ELSEVIER

Theoretical Computer Science 282 (2002) 53–99

Theoretical Computer Science

www.elsevier.com/locate/tcs

Proof assistance for real-time systems using an interactive theorem prover ☆

Paul Z. Kolano

Trusted Systems Laboratories, 303 Almaden Blvd., Suite 600, San Jose, CA 95110, USA

Abstract

This paper discusses the adaptation of the PVS theorem prover for performing analysis of real-time systems written in the ASTRAL formal specification language. Several issues arose during the encoding of ASTRAL that are relevant to the encoding of many real-time specification languages such as encoding formulas as types, handling partial functions, dealing with noninterleaved concurrency, and defining irregular operators. These issues and possible solutions are presented as well as how they were handled in the ASTRAL encoding. A translator was written that translates any ASTRAL specification into its corresponding PVS encoding. After performing the proofs of several systems using their translations, PVS strategies were developed to automate the proofs of certain types of properties. In particular, strategies are presented for fully automating the proofs of certain classes of untimed properties. In addition, strategies were developed for partially automating the derivation of timed executions using transition steps. The encoding was used as the basis for a fully automated transition sequence generator tool, which has a wide variety of applications. © 2002 Elsevier Science B.V. All rights reserved.

Keywords: Theorem proving; Real-time systems; Formal methods; ASTRAL; Proof assistance

1. Introduction

A real-time system is a system that must perform its actions within specified time bounds. With the advent of cheap processing power and increasingly sophisticated consumer demands, real-time systems have become commonplace in everything from refrigerators to automobiles. Besides such numerous everyday uses, real-time systems are also being employed in more complex and potentially deadly applications such as weapons systems and nuclear reactor controls where deviation from critical timing requirements can result in disastrous loss of lives and/or property. It is thus desirable to extensively test and verify the designs of these systems to gain assurance that such disasters will not occur. A number of formal methods for real-time systems have

☆ A preliminary version of this paper appeared in Proc. 5th AMAST Workshop on Real-Time and Probabilistic Systems, Springer, Berlin, 1999, pp. 315–333.

E-mail address: paul.kolano@trustedsyslabs.com (P.Z. Kolano)

PII: S0304-3975(01)00045-7

been proposed [18] that provide a framework under which developers can eliminate ambiguity, reason rigorously about system design, and prove that critical requirements are met using well-defined mathematical techniques.

Real-time systems are characterized by concurrency, asynchrony, nondeterminism, and dependence upon the external operating environment, which make the formal proofs of even simple real-time systems nontrivial. Even when the formal proofs of real-time systems can be performed, there is no guarantee that the proofs are valid due to flaws in reasoning caused by human error. To provide maximal assurance that the critical requirements are met, a mechanical theorem prover must be used. A mechanical theorem prover prevents flaws in reasoning by allowing proofs to proceed only in sound, well-defined steps. Besides keeping reasoning sound, theorem provers have many other benefits. They assist in the manipulation of formulas and have the ability to finish trivial subproofs automatically. Theorem provers also provide bookkeeping features such as recording the completion status of each proof. In addition, proofs can be saved, which allows them to be rerun during the maintenance phase and provides a standard proof documentation style. Finally, a theorem prover aids in the rigorous definition of a specification language by allowing its semantics to be formally defined within the language of the prover instead of using a "pencil and paper" semantics.

The use of a mechanical theorem prover also suffers from a number of drawbacks, however, that can often outweigh the benefits. In hand proofs, many details are obvious to human intuition and can be labeled "trivial" or "obvious" and not warrant further mention. In a theorem prover proof, however, these proofs must be performed explicitly and may oftentimes encompass a large number of theorem prover steps. Another drawback of theorem proving is that formulas can become unrecognizable due to either the association between the original specification language and the language of the prover or to the rewriting mechanisms of the prover's decision procedures, which can output subgoals that have nothing in common with the original goal. A related drawback is that it is sometimes difficult to examine a failed proof attempt and locate the portion of the original specification that caused the failure. This is due to either the rewriting of the decision procedures as mentioned above or to the fact that a proof in a theorem prover must often be performed in a different order and/or in a different fashion than in the corresponding proof by hand. This makes it difficult to determine what the problem was in the original specification that caused the failure.

The most significant drawback of using a mechanical theorem prover is the large number of ways in which time and effort can be wasted by performing unnecessary or repeated steps. Most of these result from a lack of careful planning such as invoking the prover while there are still many simple errors in the specification, invoking the decision procedures before enough information is present or when too much information is present, and/or choosing the ordering of the proofs or the plan of attack in an ad hoc fashion. A lack of planning can also result in duplicate proofs such as when a prover goal is split into subgoals too early.

To make theorem proving more practical, it is necessary to develop techniques to alleviate as many of these drawbacks as possible. This paper discusses two such

techniques in the context of real-time specification languages. In particular, it discusses the encoding of ASTRAL [7], which is a specification language for real-time systems, into the PVS theorem prover [10] in a way that preserves the constructs of ASTRAL as much as possible. In this way, the user can determine more easily why the proof failed in the original specification and results will be less likely to become unrecognizable since they will closely mirror the original language. During the encoding, a number of issues were encountered that are relevant to the encoding of many real-time specification languages. The different choices for handling these issues are presented as well as the decisions that were made for ASTRAL.

The other technique is to perform the proofs of a variety of different systems and encapsulate recurring proof steps into predefined automated strategies that can be applied by the user when similar situations arise. In this way, trivial proofs can stay trivial even though many complex actions may be taking place behind the scenes. Also, the possibility of splitting goals too early or invoking the decision procedures when there is too much or too little information is minimized since the strategies are written to perform these actions in the most efficient manner. The automated strategies can also be used to construct more complex tools that assist in the proof planning process. This paper discusses automated strategies for discharging obligations frequently occurring within ASTRAL proofs. In addition, a transition sequence generation tool was developed that can be used to help visualize the operation of the system when deciding on a plan of attack.

Techniques for resolving the other issues such as building a hierarchy of tools to eliminate the majority of errors in an efficient manner before the theorem prover is invoked, choosing a proof order that minimizes duplication when errors are found, and developing a high-level proof sketch that can be used as a blueprint for the associated theorem prover proof are covered in [21].

The remainder of this paper is organized as follows. In Sections 2 and 3, brief overviews of ASTRAL and PVS are given. In Section 4, the issues encountered during the encoding of ASTRAL are discussed. Section 5 describes the ASTRAL to PVS translator. Strategies for automating ASTRAL proofs and the use of PVS to develop a transition sequence generator are presented in Section 6. Section 7 discusses related work. Finally, Section 8 provides some conclusions and directions for future research.

2. ASTRAL

Many of the examples in this paper are taken from the specification of an elevator control system that was adapted from a description in [11], where an n story building is serviced by the elevator. A panel of n buttons is located inside the elevator car to request that the elevator move to a given floor. Each floor in the building also has a button panel, which has an up and a down button to request that the elevator stop at the floor and move in the corresponding direction. The elevator must service all the requests in one direction before it can move in the opposite direction. When the elevator

arrives at a floor en route to another destination and no request has been made inside the elevator for that floor, nor has a request been made at that floor's button panel for movement in the same direction, the elevator continues on to its next destination without stopping or opening the door. If such a request has been made, however, then the elevator stops and opens the door. The door is always opened for a duration of t_stop at which point it closes. When the elevator arrives at a floor that is the last request in its direction of movement, the door opens and then its behavior depends on the situation in the building. If the button panel at the elevator's location has requested movement in the same direction, the user must get in and push the desired floor on the elevator's button panel before the door has finished closing. Otherwise, the elevator is free to move in the opposite direction to service another request, if one exists. The critical timing requirement of the elevator system is that the elevator must service any request within t_service_request time of when the button was pushed.

In ASTRAL [7], a real-time system is described as a collection of state machine specifications, each of them representing a process type of which there may be multiple, statically generated, instances. There is also a *global specification*, which contains declarations for types and constants that are shared among more than one process type, as well as assumptions about the global environment and critical requirements for the whole system.

An ASTRAL *process specification* consists of a sequence of *levels*. Each level is an abstract data type view of the system being specified. The first ("top level") view is a very abstract model of what constitutes the process (types, constants, variables), what the process does (state transitions), and the critical requirements the process must meet (invariants and schedules). Lower levels are increasingly more detailed with the lowest level corresponding closely to high-level code. Fig. 1 shows one of the process types of the elevator control system. The Elevator_Button_Panel process represents the button panel located within an elevator car.

The process being specified is thought of as being in various *states*, with one state differentiated from another by the values of its *state variables*, which can be changed only by means of *state transitions*. Every process can export both state variables and transitions; as a consequence, the former are readable by other processes and the external environment while the latter are executable from the external environment. Processes communicate by broadcasting the values of exported variables and the start and end times of exported transitions. In the Elevator_Button_Panel process, the floor_requested variable and the request_floor transition are exported. The position, door_open, and door_moving variables are imported from the elevator process and a few types and constants are imported from the global specification.

Transitions are described in terms of entry and exit assertions, where *entry assertions* describe the constraints that state variables must satisfy in order for the transition to fire, and *exit assertions* describe the constraints that are fulfilled by state variables after the transition has fired. Variables are changed atomically at the end of a transition's execution with variables not referenced in the exit assertion remaining unchanged. An explicit nonnull duration is associated with each transition.

```
PROCESS Elevator_Button_Panel
  IMPORT
    floor, request_dur, clear_dur,
    elevator, elevator.position,
    elevator.door_open,
    elevator.door_moving
  EXPORT
    floor_requested, request_floor
  VARIABLE
    floor_requested(floor): boolean
  INITIAL
    FORALL f: floor
      (∼floor_requested(f))
  TRANSITION request_floor(f: floor)
    ENTRY [TIME: request_dur]
      ∼floor_requested(f)
    EXIT
      floor_requested(f)
        Becomes TRUE
TRANSITION clear_floor_request
  ENTRY [TIME: clear_dur]
    floor_requested(elevator.position)
  & ∼elevator.door_open
  & elevator.door_moving
  EXIT
    floor_requested(elevator.position)
      Becomes FALSE

ENVIRONMENT
  ( FORALL f: floor
      ( Change(floor_requested(f), now)
      & ∼floor_requested(f)
    ⟶ FORALL t: time
        ( Start(request_floor(f)) ⩽ t
        & t ⩽ now
        ⟶ ∼Call(request_floor(f), t))))
  & ( Change(elevator.door_moving, now)
    & elevator.door_moving
    & elevator.door_open
  ⟶ FORALL t: time
      ( t ⩾ Change2 (elevator.door_moving)
      ⟶ ∼Call(request_floor(
              elevator.position), t)))
INVARIANT
  FORALL f: floor
    ( Change(floor_requested(f), now)
    & ∼floor_requested(f)
  ⟶ EXISTS t: time
      ( Change2 (floor_requested(f)) < t
      & t ⩽ now
      & past(elevator.position, t)=f
      & ∼ past(elevator.door_open, t)
      & past(elevator.door_moving, t)))
```

Fig. 1. The Elevator_Button_Panel process.

Each transition is either a local transition or an exported transition. A local transition is enabled when its entry assertion is satisfied. An exported transition, however, is only enabled when both its entry assertion is satisfied and when it has been called (i.e. invoked) from the external environment. Transitions are executed as soon as they are enabled assuming no other transition for that process instance is executing. If two or more transitions are enabled simultaneously, a nondeterministic choice will occur and only one of them will execute. In the Elevator_Button_Panel process, the clear_floor_request transition is enabled when the elevator is currently stopped with its door opening at a floor that has been requested but not yet serviced.

In addition to specifying system state (through process variables and constants) and system evolution (through transitions), an ASTRAL specification also defines system critical requirements and assumptions about the behavior of the environment that interacts with the system. The behavior of the environment is expressed by means of *environment clauses*, which describe assumptions about the pattern of invocation of external transitions. ASTRAL also allows assumptions about the context provided by other processes in the system to be expressed in the *imported variable clause*. This clause describes patterns of changes to the values of imported variables, including

timing information about transitions exported by other processes that may be used by the process being specified. Critical requirements are expressed by means of *invariants* and *schedules*. Invariants represent requirements that must hold in every state reachable from the initial state, no matter what the behavior of the external environment is, while schedules represent additional properties that must be satisfied provided that the external environment and the other processes behave as assumed.

The requirement and assumption clauses are expressed using a combination of first-order logic and ASTRAL-specific constructs. The main constructs are the timed operators used to express timing requirements. The *start* operator, Start(trans1, t1), takes a transition trans1 and a time t1 and returns true iff the last start of trans1 was at t1. Similarly, the *end* and *call* operators, End(trans1, t1) and Call(trans1, t1), return true iff the last end or the last call of trans1 was at t1. The *change* operator, Change(A, t), takes an expression A and a time t and returns true iff the last time A changed value was at t. The *past* operator, past(A, t), takes an expression A and a time t and returns the value of A at t. In addition to these operators, a special global variable *now* is used to denote the current time, where the time domain is the nonnegative real numbers.

Using these operators, a variety of complex properties can be expressed. For example, the invariant of the Elevator_Button_Panel process states that between a change to floor_requested(f) and a change back to $\sim$floor_requested(f) for any floor f, the elevator has been at f and its door has started opening. The first portion of the environment states that any pushes to the button for floor f should be ignored when floor_requested(f) is already true. The second portion states that requests cannot be made of the elevator to stop at a floor between when the door starts opening on that floor until when it starts closing. Note that both the invariant and environment use the operator $\text{Change}_2(A, t)$, which is true iff the second change in the past to the value of expression A occurred at time t. An introduction and complete overview of the ASTRAL language can be found in [7]. For the interested reader, the complete specification of the elevator system is given in the appendix.

Rather than implementing a theorem prover for ASTRAL from scratch, it was decided to take advantage of an existing general-purpose theorem prover adapted for use with ASTRAL. PVS was considered ideal for ASTRAL given its powerful typing system, higher-order facilities, heavily automated decision procedures, and ease of use. Other theorem provers were also considered, including HOL [15] and ACL2 [19]. HOL does not have the usability of PVS and its decision procedures are not as powerful [14]. ACL2 is also not as usable as PVS and has limited or no support for arbitrary quantification and real numbers [31] which are both required for ASTRAL.

3. PVS

The prototype verification system (PVS) [10] is a powerful interactive theorem prover based on typed higher-order logic. A PVS specification consists of a modular collection of *theories*. A theory may be parameterized to support polymorphism.

Declarations in one theory can be referenced in another theory by using an *importing clause*. Parameterized theories can be imported either with explicit parameters or without parameters. If left without parameters, PVS attempts to instantiate the theory based on the use of its declarations within the importing theory. Most single parameter theories can be instantiated automatically by PVS, but theories with complex or multiple parameters often need to be instantiated explicitly in the referring theory.

The PVS language is very flexible and expressions can contain arbitrary quantification, recursion, and higher-order constructs. With these facilities, PVS can specify most, if not all, statements of higher-level programming languages. For example, the *mapcar* function in Lisp can be expressed as the following.

```
mapcar(f: [T1 → T2], l: list[T1]): RECURSIVE list[T2] =
    CASES l OF
        null: null,
        cons(x, y): cons(f(x), mapcar(f, y))
    ENDCASES
MEASURE (LAMBDA (f: [T1 → T2], l: list[T1]): length(l))
```

The *measure* at the end of the mapcar definition must be given in every recursive PVS function definition. It has the same signature as the associated function and defines an expression that decreases in each recursive iteration, which is used to prove the termination of the function.

A PVS theory declaration consists of a set of types, constants, axioms, and theorems. PVS has a very expressive typing language, which includes functions, arrays, sets, tuples, enumerated types, and predicate subtypes. Types may be *interpreted* or *uninterpreted.* Interpreted types are defined based on existing types, while uninterpreted types must be defined axiomatically. Predicate subtypes allow the expression of complex types that must satisfy a given constraint. For example, the even numbers can be defined as shown below.

```
even_int: TYPE = {i: int | (EXISTS (j: int): 2 * j = i)}
```

For any assignment or substitution that involves a predicate subtype, PVS generates *type correctness conditions* (TCCs), which are obligations that must be proved in order for the rest of the proof to be valid. For example, consider the following declaration.

```
e_plus_2(e: even_int): even_int = e + 2
```

PVS generates the TCC shown below for the definition of e_plus_2.

```
% Subtype TCC generated (line 7) for e + 2
        e_plus_2_TCC1: OBLIGATION
          (FORALL (e: even_int): (EXISTS (j: int): 2 * j = e + 2))
```

That is, it must be shown that adding two to an even number is still an even number. Otherwise, the definition of e_plus_2 violates its stated type.

Like types, constants can either be interpreted or uninterpreted. The value of an interpreted constant is stated explicitly, whereas the value of an uninterpreted constant

is defined axiomatically. For example, the definition of push in

```
stack: TYPE = list[T];
push(e: T, s: stack): stack = cons(e, s);
```

is an interpreted constant, because the exact effect of a push statement can be determined by expanding its definition. The definition of push in

```
stack: TYPE;
push: [[T, stack] → stack];
```

is uninterpreted because all that is known about push is that applying it to a tuple of type $[T, \text{stack}]$ returns a stack of unknown content. In the former definition, the exact consequence of the push operation is given in terms of list operations. To express properties about an uninterpreted constant, however, axioms must be used. For example, in the previous declaration, the following would be appropriate:

```
top_of_push: AXIOM
    top(push(e, s)) = e
```

This states that no matter how stack, push, and top are implemented, applying top to the stack resulting from a push operation will result in the element just pushed. In general, axioms describe anything that is considered to be a "truth" in a theory. Besides types, constants, and axioms, the other basic component of a theory are theorems, which are hypotheses that are thought to be true, but that need to be proven with the help of the prover.

When the PVS prover is invoked on a theorem, the theorem is displayed in the form of a *sequent*. A sequent consists of a set of *antecedents* and a set of *consequents*, where if $A_1, \ldots, A_n$ are antecedents and $C_1, \ldots, C_n$ are consequents in the current sequent, then the current goal is $(A_1 \& \ldots \& A_n) \rightarrow (C_1 | \ldots | C_n)$. It is the user's job to direct PVS with prover commands such as instantiating quantifiers and introducing lemmas to show that either (1) there exists an i such that A_i is false, (2) there exists an i such that C_i is true, or (3) there exists a pair (i, j) such that $A_i = C_j$. PVS maintains a proof tree, which consists of all of the subgoals generated during a proof. Initially, when the prover is invoked on a theorem, the proof tree contains only the sequent form of that theorem. As the proof proceeds, subgoals may be generated and proved. To prove that a particular goal in the proof tree holds, all of its subgoals must be proved true. PVS allows the user to define *strategies*, which are collections of prover commands that can be used to automate frequently occurring proof patterns.

4. Encoding issues

While encoding ASTRAL within PVS, a number of issues arose that needed to be handled. Several of these issues are not exclusive to ASTRAL and occur in many different real-time specification languages. The following sections discuss some of these issues and how they were handled in the ASTRAL encoding.

4.1. Formulas as types

In many real-time specification languages, a single formula may have multiple values depending on the temporal context in which it is evaluated. Depending on the language, the temporal context may be an explicit clock variable, or implicitly derivable from the formula. To encode such languages into a theorem prover, it is necessary to define formulas as types that can be evaluated in different contexts.

Two different approaches have been used to encode formulas as types in PVS. In the TRIO to PVS encoding [1], an uninterpreted "TRIO_formula" type is introduced to handle this issue. In TRIO, the current time is always implicit, but the values of formulas in the past and future can be obtained relative to the current time using the *dist* operator, dist(A,t), which takes a formula A and a relative time t and gives the value of A at t time units from the current time. In the TRIO encoding, the dist operator is defined as a function of type [[TRIO_formula, time] → TRIO_formula]. Axioms are defined to transform elements of type TRIO_formula to other elements of type TRIO_formula. Eventually, there must be a valuation from TRIO_formulas to real-world values (i.e. booleans, integers, etc.) so that the decision procedures of PVS can be invoked. Hence, a valuation function is defined that takes a TRIO_formula and produces the corresponding boolean value assuming an initial context of the current time instant.

The duration calculus (DC) is another real-time language that has been encoded into PVS [28]. DC is an implicit-time interval temporal logic in which the current interval is not explicitly known. Rather than using uninterpreted types to define formulas, however, the DC encoding takes advantage of the higher-order capabilities of PVS and defines formulas as functions of type [Interval → bool]. DC operators are defined as Curried functions, which when given their original operands, return a function from an Interval to the original range of the operator. For example, the disjunction operator "$\bigvee$" is defined as "$\bigvee(A,B)(i)$: bool $= A(i)$ OR $B(i)$", where A and B are of the type [Interval → bool] and i is of type Interval. Using this technique, the resulting functions can be combined normally, while still delaying the evaluation of the whole expression until a temporal context is given. Eventually, when a specific interval is given, an actual boolean value is obtained.

For ASTRAL, the DC approach was chosen for several reasons. Since TRIO is an implicit-time temporal logic, one of the main motivations of the TRIO encoding was to keep the actual current time hidden. In ASTRAL, the current time can be explicitly referenced using the variable now, thus it was unnecessary to keep the time hidden. Another disadvantage of the TRIO encoding is that all of the axioms of first-order logic needed to be explicitly encoded into PVS to manipulate the TRIO_formula type. Using the DC encoding style, however, the built-in PVS framework could be utilized, which includes all first-order logic axioms.

All ASTRAL operators have been defined as Curried functions from their operand domains to the type [time → range]. For example, the ASTRAL operator Start(trans1, t1) takes a transition trans1 and a time t1 and returns true iff the last start of trans1 was at t1. Its PVS counterpart, Start1(trans1, at1) takes a transition trans1 and an operand

at1 of type [time → time] and returns a function of type [time → bool] such that when an evaluation time t1 is given will return true iff the last start of trans1 at time $t1$ was at time at1(t1). In the Start1 definition, shown below, as well as the definitions of all ASTRAL operators that take a time operand, the time operand is itself of type [time → time] and is only evaluated after an evaluation context is provided.

```
Start1(trans1: transition, at1: [time → time]) (t1: {t1: time | at1(t1) ⩽ t1}): bool =
    Fired(trans1, at1(t1)) AND
    (FORALL (t2: time):
        at1(t1) < t2 AND t2 ⩽ t1 IMPLIES
        NOT Fired(trans1, t2))
```

With the operators defined in this manner, it is possible to combine ASTRAL operators in standard ways and yet still produce an expression that will only be evaluated once its temporal context is given. The explicit operator definitions also allow all expressions translated from ASTRAL to PVS to be easily expanded and reduced via the built-in mechanisms of PVS. The resulting encoding is very close to the ASTRAL base logic with only slight syntactic differences and allows a specifier who is familiar with the ASTRAL language to easily read the PVS expressions of ASTRAL formulas.

4.2. Partial functions

Some specification languages such as Z [29] allow the definition of partial functions (i.e. functions that are only well defined at certain points) within specifications. Unlike some other theorem provers, PVS does not support the use of partial functions directly. To encode languages that allow the definition of partial functions or whose operators themselves may be partial functions into PVS, alternative approaches must be used. In lieu of partial functions, PVS has a very powerful predicate subtyping system that allows functions to be declared with domains of only those elements satisfying a given predicate, such as only those elements for which a function is well defined. The user then proves TCC obligations that the operand of each function satisfies the given predicate. For a specific class of functions, such as boolean functions, an alternative to predicate subtyping is to define a new domain that contains an additional undefined element and then modify the operators for that class of functions to use the new domain. For example, for boolean partial functions, a three-valued domain of {true, false, undefined} can be defined in PVS with boolean operators modified to work with the new domain.

The partial functions in ASTRAL are the operators that take a time as an argument. In ASTRAL, only times in the past may be referenced, thus any formula that references a time beyond the value of now is undefined. In encoding these operators into PVS, the choice was made to use the subtyping mechanism of PVS for similar reasons as the choice to use the DC encoding style. Namely, it was preferable to rely on the existing PVS framework as much as possible. There were also a number of disadvantages to explicitly adding an undefined value and then modifying the appropriate operators. For instance, many additional axioms needed to be added to derive and manipulate

expressions containing the undefined element. The main drawback, however, is that the ASTRAL past operator is a polymorphic function. That is, the past operator can have multiple types depending on the type of A. Since past takes a time, it is undefined when t is greater than now. Since A can be of any type, essentially every type in the specification and hence every operator in the language would need to be redefined using an undefined element. This was highly undesirable and would have unnecessarily complicated both the encoding and the resulting proofs.

Instead, by using the PVS subtyping mechanism, the user must prove TCCs showing that the time operand of any timed operator used in a specification is less than or equal to the temporal context given to the operator. Most of these obligations will be trivial given that the time operands are usually based on now directly or on a quantified time variable that was appropriately limited.

The definition of the Start1 operator in the previous section demonstrates the use of the subtyping mechanism. The time operand of the Start1 function, at1, is of type [time → time] and is only evaluated after an evaluation context is provided. Since it is not known whether at1(t1) will be a valid operand or not (i.e. will cause the expression to be undefined), t1 is limited by the PVS typing system to be greater than or equal to at1(t1). It is then the user's job to show via a TCC obligation that any evaluation times of a Start1 expression occurring in a specification are permissible. The other timed operators of ASTRAL are defined in a similar manner.

4.3. Noninterleaved concurrency

Concurrency in real-time systems can be represented by either an interleaved or a noninterleaved model. In an interleaved model, concurrent events occur sequentially between changes to time, while in a noninterleaved model, concurrent events occur simultaneously without an implied ordering. Timed state-machine languages that use an interleaved model of concurrency use an explicit "tick" transition to change time. The combination of the implied ordering of interleaved concurrency and the use of a tick transition allows the semantics of interleaved timed state-machine languages to be simplified significantly over their noninterleaved counterparts because a system execution can be represented as a sequence of transitions rather than an interval of time in which one or more events occur or do not occur at each time. The proof obligations for such languages are also correspondingly simplified since they can be inductive on the nth transition to fire rather than a full induction on a possibly dense time domain.

In ASTRAL, the proof obligations are carried out modularly by proving the properties of each process individually and then proving global properties based on the collection of process properties. Fig. 2 shows the dependencies of the proof obligations in the elevator control system. In this figure, the requirements of the Elevator_Button_Panel (EBP) are proved using its actual executions as well as the behavior it assumes about the external environment in its environment clause. The requirements of the Floor_Button_Panel (FBP) are proved similarly. The requirements of the Elevator process are proved using its actual executions, but do not require any assumptions

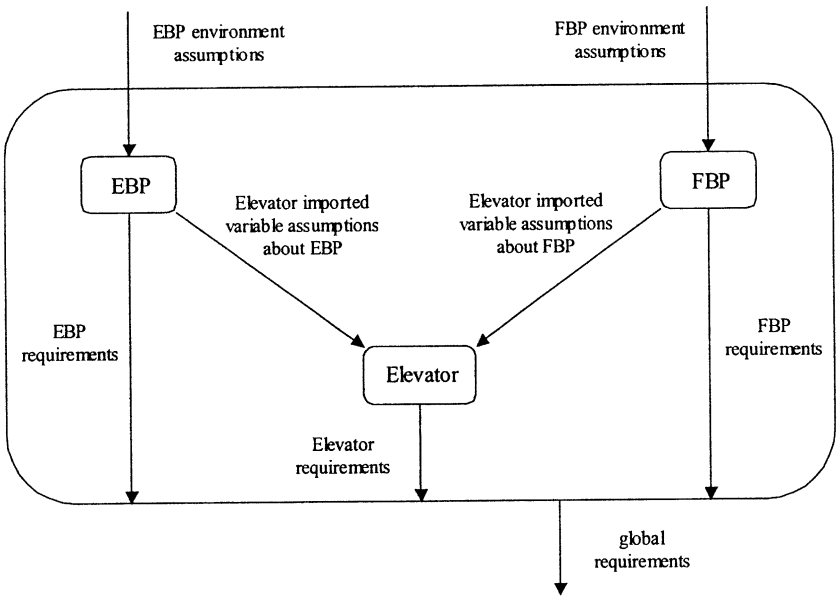

Fig. 2. Proof dependencies of elevator control system.

about the external environment. Instead, they require assumptions about the behavior of the Elevator_Button_Panel and Floor_Button_Panel types as stated in the Elevator's imported variable clause. Note that the Elevator does not depend on the actual executions of the other two process types, but only depends on the behavior that is assumed about these processes. The global requirements do not depend on any actual executions and are dependent solely on the local requirements of each process (and the global environment, when present).

Although the proof obligations of a process are proved using assumptions about other processes rather than the actual executions of those processes, these assumptions in essence define subsets of actual executions by restricting imported variable values and the times at which imported transition may start and end. Thus, although each process's execution is a sequential series of transitions (with varying delays in between) by the fact that transitions are nonoverlapping on each process instance, the events in other processes represented by the "assumption executions" overlap with each other and with events in the local process, thereby requiring a noninterleaved model.

4.3.1. ASTRAL axiomatization

The axiomatization of ASTRAL into PVS is a much revised and expanded version of the ASTRAL axiomatization of [8] and includes corrections for both soundness and completeness. The full version of the semantics presented in this paper defines the current formal semantics of the ASTRAL language. The ASTRAL axiomatization is defined by three types of axioms. The abstract machine axioms describe the execution

```
trans_fire: AXIOM
    (FORALL (t1: time):
       (EXISTS (trans1: transition):
          Enabled(trans1, t1)) AND
       (FORALL (trans2: transition, t2: time):
          t1 - Duration(trans2) < t2 AND t2 < t1 IMPLIES
             NOT Fired(trans2, t2)) IMPLIES
          (EXISTS (trans1: transition): Fired(trans1, t1)))
trans_entry: AXIOM
    (FORALL (trans1: transition, t1: time):
       Fired(trans1, t1) IMPLIES
          Entry(trans1, t1))
trans_exit: AXIOM
    (FORALL (trans1: transition, t1: time):
       t1 ⩾ Duration(trans1) AND
       Fired(trans1, t1 - Duration(trans1)) IMPLIES
          Exit(trans1, t1))
trans_called: AXIOM
    (FORALL (trans1: transition, t1: time):
       Fired(trans1, t1) AND
       Exported(trans1) IMPLIES
          Issued_Call(trans1, t1))
```

```
trans_mutex: AXIOM
    (FORALL (trans1: transition, t1: time):
       Fired(trans1, t1) IMPLIES
          (FORALL (trans2: transition):
             trans2 ≠ trans1 IMPLIES
                NOT Fired(trans2, t1)) AND
          (FORALL (trans2: transition, t2: time):
             t1 < t2 AND
             t2 < t1 + Duration(trans1) IMPLIES
                NOT Fired(trans2, t2)))
vars_no_change: AXIOM
    (FORALL (t1: time, t3: time):
       t1 ⩽ t3 AND
       (FORALL (trans2: transition, t2: time):
          t1 < t2 + Duration(trans2) AND
          t2 + Duration(trans2) ⩽ t3 IMPLIES
             NOT Fired(trans2, t2)) IMPLIES
          (FORALL (t2: time):
             t1 ⩽ t2 AND t2 ⩽ t3 IMPLIES
                Vars_No_Change(t1, t2)))
initial_state: AXIOM
    Initial(0)
```

Fig. 3. ASTRAL abstract machine axioms.

of a single process. The imported transition axioms describe information that can be derived about the execution of other processes. Finally, the specification-dependent axioms, which will not be discussed, are axioms that can only be constructed after a specification is given.

4.3.1.1. Abstract machine axioms

The seven ASTRAL abstract machine axioms are shown in Fig. 3. The axioms are based on the predicates *Called* and *Fired*. Called(trans1, t1) is true iff transition trans1 was called from the external environment at time t1. Fired(trans1, t1) is true iff trans1 fired at t1. Since a different transition may be executing on each process instance, each process instance has a separate Fired and Called predicate. In ASTRAL, a given process instance "knows" its own execution history completely, but only knows the portion of the execution history of other process instances that pertains to the exported transitions of those instances. In the semantics, for a given process instance, the Fired and Called predicates of the process can be used to derive information about the state variables of the process and vice versa. The predicates of other process instances, however, can only be used to derive a limited amount of information as will be discussed in the next section.

The *trans_fire* axiom is the only way to directly derive that a transition fired. It states that if some transition is enabled and the process is idle (i.e. no other transition is in the middle of execution), then some transition will fire. Note that Enabled requires

that the transition's entry assertion holds and that if the transition is exported, then it has been called.

The trans_fire axiom by itself is not sufficient to describe what occurs when a transition fires. A number of other axioms make assertions that further describe the behavior of a process. The *trans_entry* axiom states that whenever a transition fires, its entry assertion held at that time.

The *trans_exit* axiom states that whenever a transition fires, its exit assertion holds at a time duration later. Note that in this case, the user must guarantee that the exit assertion will not evaluate to false for the axiom to be sound. In the case of trans_entry, this requirement is not necessary because it is not possible to derive Fired(trans1, t1) if Entry(trans1, t1) does not hold. In the trans_exit case, however, it is possible to derive Fired(trans1, t1), regardless of the value of Exit(trans1, t1 + Duration(trans1)).

The *trans_called* axiom states that whenever an exported transition fires, it must have been called since the last time the transition fired.

The *trans_mutex* axiom states that whenever a transition fires, no other transition can fire until duration later (i.e. until the transition ends). This axiom combined with trans_fire is sufficient to show that a single unique transition fires when some transition is enabled and the process is idle. Note that since the semantics cannot be represented by a sequence of transitions as in an interleaved model, it is necessary to assure that a process is actually idle in order for a transition to fire.

These six axioms describe the dynamic execution of transitions. Besides the start, end, and call times of transitions, the other time-dependent entities are variables. The axioms so far only describe variables implicitly in the Entry, Exit, and Enabled functions used in them. Thus, the value of a variable is only known at the time a transition starts and when it ends. In ASTRAL, however, it is also known that a variable only changes value when a transition ends. Thus, the *vars_no_change* axiom states this fact. Specifically, it states that for any interval in which a transition has not ended, all variables keep a single value throughout the interval. The Vars_No_Change function is process-dependent and is constructed by the translator based on the variables declared in each process. Vars_No_Change(t1, t2) states that the value of all variables of the process have the same value at t1 as they do at t2.

Finally, the *initial_state* axiom states that the initial condition holds at time zero. As was the case in trans_exit with Exit, Initial is required to be true at time zero, or else the soundness of the axiom cannot be guaranteed.

4.3.1.2. Imported transition axioms

In addition to the abstract machine axioms, there are three axioms dealing with imported transitions, which are shown in Fig. 4. Most of the information that can be derived about local variables and transitions cannot be derived about imported variables and transitions. For example, it is not known when imported variables will change, nor what the duration of an imported transition is, nor what held when an imported transition started or ended, etc. If any of these items are required to hold to prove a

```
i_trans_mutex: AXIOM
    (FORALL (id1: id, itr1: i_transition, t1: time, t3: time):
        t1 < t3 AND
        i_Started(id1, itr1, t1) AND
        (FORALL (t2: time):
            t1 < t2 AND t2 ⩽ t3 IMPLIES
                NOT i_Ended(id1, itr1, t2)) IMPLIES
            (FORALL (itr2: i_transition, t2: time):
                t1 < t2 AND t2 ⩽ t3 IMPLIES
                    NOT i_Started(id1, itr2, t2) AND
                    NOT i_Ended(id1, itr2, t2)))
```

```
i_trans_end: AXIOM
    (FORALL (id1: id, itr1: i_transition, t3: time):
        i_Ended(id1, itr1, t3) IMPLIES
            (FORALL (itr2: i_transition):
                itr2 ≠ itr1 IMPLIES
                    NOT i_Ended(id1, itr2, t3)) AND
            (EXISTS (t1: time):
                t1 < t3 AND
                i_Started(id1, itr1, t1) AND
                (FORALL (t2: time):
                    t1 < t2 AND t2 < t3 IMPLIES
                        NOT i_Ended(id1, itr1, t2))))
i_trans_start: AXIOM
    (FORALL (id1: id, itr1: i_transition, t3: time):
        i_Started(id1, itr1, t3) IMPLIES
            (FORALL (itr2: i_transition):
                itr2 ≠ itr1 IMPLIES
                    NOT i_Started(id1, itr2, t3)) AND
            (EXISTS (t1: time):
                t1 ⩽ t3 AND
                i_Called(id1, itr1, t1) AND
                (FORALL (t2: time):
                    t1 ⩽ t2 AND t2 < t3 IMPLIES
                        NOT i_Started(id1, itr1, t2))))
```

Fig. 4. ASTRAL imported transition axioms.

requirement, they must be explicitly stated in an imported variable clause. There are, however, a few things that can be deduced about all imported transitions, regardless of context.

The imported axioms are expressed in terms of *i_Called*, *i_Started*, and *i_Ended*, which are shown below. These functions correspond to the local definitions of Called and Fired, but refer to information about transitions imported from other processes. These predicates represent the assumed executions discussed above. Namely, they are defined by the imported variable clause of the process being reasoned about. The id parameter defines the process instance for which the predicates are defined. The exact duration between a start and an end of an imported transition is not known globally or in other processes because the duration is implementation dependent. Thus, i_Started and i_Ended had to be defined separately, rather than the single Fired of local process definitions.

```
i_Started: [[id, i_transition, time] → bool]
i_Ended: [[id, i_transition, time] → bool]
i_Called: [[id, i_transition, time] → bool]
```

Based on these definitions, three axioms can be defined that hold for all transitions in imported processes regardless of the imported variable clause of the process being

reasoned about. The *i_trans_mutex* axiom states that for any process id and in any interval such that an imported transition started at the beginning of the interval and has not yet ended, no imported transition can have started or ended on the process associated with that process id within the interval (excluding the first instant).

The *i_trans_end* axiom states that for any process id, if an imported transition has ended on that process, no other imported transition ended on the same process at the same time and there was a start that has occurred since the last time the transition ended.

The *i_trans_start* axiom is similar to i_trans_end, except that it states that if an imported transition starts, then no other imported transition started on the same process at the same time and that the transition has been called but not yet serviced.

4.3.3. Proof obligations

Since ASTRAL is based on noninterleaved concurrency, the intra-level proof obligations [8] (i.e. the proof obligations necessary to show that the invariant and schedule of a level hold) are inductive on ASTRAL's time domain. Since the time domain of ASTRAL is the nonnegative real numbers, however, and simple induction on that domain is not valid, the induction must be performed on nonempty intervals of the nonnegative reals. That is, the induction hypothesis is assumed up to some arbitrary time $T0$ and the user must show that it holds for a constant length of time $\Delta > 0$ afterwards. Note that Δ is an arbitrary constant and can take any value as long as it is positive. The induction case of the invariant proof obligation is shown below.

```
invariant_induct: THEOREM
  (FORALL (T1: time): T1≤T0 IMPLIES Invariant(T1)) IMPLIES
    (FORALL (T1: time): T0<T1 AND T1<T0 + Δ IMPLIES Invariant(T1))
```

For the induction to be reasonable, Δ must be bounded because the bigger Δ becomes, the more difficult it is to prove that the property holds at the times close to the upper bound $T0 + \Delta$. This is because at those times, more and more time has elapsed since the last known state of the system (i.e. when the inductive hypothesis held). In translating the proof obligations into PVS, it was not possible to say that Δ is "as small as possible". Instead, an explicit upper bound needed to be chosen to restrict Δ. The upper bound chosen for the ASTRAL encoding was a value less than the smallest transition duration. That is, the conjunct "(FORALL (trans1: transition): $\Delta <$ Duration(trans1))" was added to the proof obligation above.

This bound is satisfactory for a number of reasons. The main justification is that with Δ bounded by the smallest duration, only a single transition can fire or complete execution within the proof interval. This is advantageous because if only a single transition can end, then the state variables can only change once within the interval. Additionally, if a transition did end within the interval, then the inductive hypothesis held when the transition began firing. These qualities are useful for automating the proofs of certain types of properties as will be shown in Section 6.1.

4.4. Irregular operators

In some specification languages, there are operators whose type signatures cannot be described in a regular fashion. One example is the ASTRAL Start operator. For unparameterized transitions, the signature of the Start operator is regular and can be written as "[transition, time] → boolean". For parameterized transitions, however, the transition operand can also be a transition name with a parameter list. For a transition trans1 with n parameters of arbitrary type $(p_1, \ldots, p_n)$, all of the following are legal ASTRAL expressions: Start(trans1, t1), Start(trans1(p_1), t1), Start(trans1(p_1, p_2), t1), ..., Start (trans1($p_1, \ldots, p_n$), t1). Since the parameters are of arbitrary and possibly differing types, there is no type signature that can adequately describe the Start operator.

These "irregular" operators are difficult to encode in an elegant fashion. To encode the parameterized version of the Start operator, there seemed to be two possible alternatives. The first option was to define an overloaded Start operator for each allowable transition/parameter combination. For example, the trans1 transition above would have n corresponding Start definitions for the n possible parameter combinations. Although this would keep the PVS encoding similar to its ASTRAL counterpart, it would also increase the size of translated specifications significantly. In addition, it was undesirable to define the core ASTRAL operators in translated specifications rather than in a standard ASTRAL–PVS library. Instead, a second option was chosen, which was to define a single parameter type that contains fields for every possible parameter combination. This allowed a single Start definition to handle all of the parameter cases.

In general, a timed irregular operator can be "regularized" in four steps. First, a general parameter type is created that covers all parameter combinations. Then, a history of parameters is set up to record the parameters used at different times in the system's execution. Next, a parameter evaluation function is defined to test the equality of two parameter instances for a given operand (e.g. transition). Finally, the operator is redefined appropriately.

4.4.1. Defining a parameter type

The first step is to introduce a new "parameter type" using a record declaration, which contains the parameter names and types of all transitions in the process. For example, the definition of parameter in the Elevator_Button_Panel process is shown below.

```
parameter: TYPE = [# p_floor_1: floor #]
```

The idea of this scheme is that all entry/exit assertions and transition operator definitions can reference the same type (i.e. parameter) and use only those parts of a parameter instance appropriate in the given situation. The parts of a parameter that are not used in an expression for all intents and purposes do not exist for that expression. For example, an entry assertion may reference parameters that are passed to it when called from the external environment. The entry assertion only references its own declarations within the parameter type, thus only constrains those portions of the

parameter. The unreferenced elements of the parameter type can have any value, thus they do not affect the reasoning.

4.4.2. Defining a parameter history

After the parameter type is created, it is necessary to set up a history of parameters to record the parameter instances that are used at each time in the system's execution. In ASTRAL, any transition may have parameters that are used in the entry and exit assertions to describe the conditions of enablement and the effects of execution, respectively. For an exported transition, the parameters are provided by the external environment when the transition is called. These transitions are enabled if there is a set of such parameters that has not yet been serviced by a previous execution of the transition and for which the entry assertion is satisfied. Transitions that are not exported are enabled if there is any set of parameters of the appropriate types that satisfy the entry assertion. When a parameterized transition fires, one set of the possible sets of parameters is chosen nondeterministically. In the semantics, the functions *Call_Parms* and *Fire_Parms*, shown below, are defined to record the history of transition parameters.

```
Call_Parms: [[ptrans1: {trans1: transition | Exported(trans1) AND
   Has_Parms(trans1)}, {t1: time | Called(ptrans1, t1)}] → set[parameter]]
Fire_Parms: [[ptrans1: {trans1: transition | Has_Parms(trans1)},
   {t1: time | Fired(ptrans1, t1)}] → parameter]
```

Call_Parms is only valid at times when an exported transition has been called and holds the parameters supplied by the external environment. Fire_Parms is only valid at times when a parameterized transition has fired and holds the instance of the parameters for which the transition fired. An additional requirement between Call_Parms and Fire_Parms is that if an exported parameterized transition trans1 fires at t1, the parameters for which trans1 fired must come from the set of trans1 call parameters that have not yet been serviced at t1. The *call_fire_parms* axiom describes this relationship between Call_Parms and Fire_Parms.

```
call_fire_parms: AXIOM
    (FORALL (trans1: transition, t3: time):
        Exported(trans1) AND
        Has_Parms(trans1) AND
        Fired(trans1, t3) IMPLIES
            (EXISTS (t1: time):
                t1≤t3 AND
                Called(trans1, t1) AND
                member(Fire_Parms(trans1, t3), Call_Parms(trans1, t1)) AND
                (FORALL (t2: time):
                    t1≤t2 AND t2<t3 AND
                    Fired(trans1, t2) IMPLIES
                    Fire_Parms(trans1, t2) ≠ Fire_Parms(trans1, t3))))
```

4.4.3. Defining a parameter evaluation function

Start(trans1($p_1,\ldots,p_i$), t1) is true iff the last time trans1 fired with its first i parameters equal to $p_1,\ldots,p_i$ was at time t1. The last component necessary to regularize the definition of the Start operator is a function to determine the equality of the first i parameters of a given transition in two instances of the parameter type. For each process specification, an *Eval_Parms* function is constructed with the required functionality. The Eval_Parms function of the Elevator_Button_Panel process is shown below. Eval_Parms is defined recursively on the number of parameters to check. Depending on the transition given, a different set of components of the parameter record is checked. In the definition below, the "p_floor_1" component is checked in the request_floor transition.

```
Eval_Parms(PTRANS1: {TRANS1: transition | Has_Parms(TRANS1)},
        N1: nat, P1: parameter, P2: parameter): RECURSIVE bool =
    (IF N1 = 0 THEN TRUE
    ELSE
        CASES PTRANS1 OF
            request_floor:
                IF N1 = 1 THEN p_floor_1(P1) = p_floor_1(P2)
                ELSE FALSE
                ENDIF
            ELSE FALSE
        ENDCASES AND
        Eval_Parms(PTRANS1, N1 - 1, P1, P2)
    ENDIF)
    MEASURE (LAMBDA (TRANS1: transition, N1: nat,
                P1: parameter, P2: parameter): N1)
```

4.4.4. Defining the irregular operator

With the above definitions, it is possible to provide a regular definition of the Start operator. The Start1 definition shown below is similar to the Start1 definition in Section 4.1 except that it takes a natural number n1 and a Curried parameter ap1. This definition requires that ptrans1 has fired and that the first n1 parameters of ptrans1 in ap1(t1) match the first n1 parameters of the actual fire parameters at that time. In addition, any time after the given time (at1(t1)) at which an exception associated with ptrans1 fired, the first n1 parameters must not match.

```
Start1(ptrans1: {trans1: transition | Has_Parms(trans1)}, n1: nat,
        ap1: [time → parameter], at1: [time → time])(t1: {t1: time | at1(t1) ⩽ t1}): bool =
    Fired(ptrans1, at1(t1)) AND
    Eval_Parms(ptrans1, n1, ap1(t1), Fire_Parms(ptrans1, at1(t1))) AND
    (FORALL (t2: time):
        at1(t1) < t2 AND t2 ⩽ t1 AND
        Fired(ptrans1, t2) IMPLIES
        NOT Eval_Parms(ptrans1, n1, ap1(t1), Fire_Parms(ptrans1, t2)))
```

5. PVS library and translator

The axiomatization and operator definitions discussed in Section 4 have been incorporated into an ASTRAL–PVS library. The library contains the specification-independent core of the ASTRAL language. In the axiomatization and operators, some of the theories are parameterized by type and function constants. For example, to define the trans_fire axiom, the type "transition" and the function "Duration" need to be supplied to the axiomatization. In order to use the axiomatization, the appropriate types and functions must be defined based on the specification to be verified. An ASTRAL to PVS translator has been developed to automatically construct all the appropriate definitions.

The major obstacle in translating ASTRAL specifications is translating identifiers with types involving lists and structures. In ASTRAL, it is possible to define arbitrary combinations of structures and lists as types, thus references to variables of these types can become quite complex. For example, consider the following type declarations: "list1: list of integer" and "struct1: structure of (l_one(integer): list1)". If s1 is a variable of type struct1, valid uses of s1 would include s1 by itself, s1[l_one(5)], and s1[l_one(5)][9]. The translation of expressions such as these must result in a Curried time function, so that it can be used with the definitions of the Curried boolean and arithmetic operators. The expression in each bracket can be time-dependent, so it is necessary to define the translation such that an evaluation context (i.e. time) given to the expression as a whole is propagated to all expressions in brackets.

In the translation of this example, s1 is a function of type [time $\rightarrow$ struct1] and struct1 is a record [# l_one: [integer $\rightarrow$ list1] #]. The expression "s1[l_one(5)][9]", becomes "($\lambda(T1$: time): nth((($\lambda(T1$: time): l_one((s1)($T1$)) ((const(5))($T1$))))($T1$), (const(9))($T1$)))". The lambdas are added to propagate the temporal context given to the formula as a whole. Although the lambda expression generated for s1 looks very difficult to decipher, translated expressions will never actually be used in this "raw" form. In the proof obligations, a translated expression is always evaluated in some context before being used. Once this evaluation occurs, all the lambdas drop out and the expression is simplified to a combination of variables and predicates. For example, the expression above evaluated at time t becomes "nth(l_one((s1)(t))(5), 9)". First, the value of the variable s1 is evaluated at time t. Then, the record member l_one is obtained from the resulting record. This member is parameterized, so it is given a parameter of 5. Finally, element 9 of the resulting list is obtained.

For the full details of the axiomatization of the ASTRAL abstract machine, the operator definitions, and the ASTRAL to PVS translator, see [21].

6. Proof assistance and automation

After a specification is translated, the user must prove the inductive proof obligations discussed in Section 4.3. In general, the proof obligations are undecidable so

they require a fair amount of interaction with the prover. For timed properties, this interaction usually consists of setting up the sequences of transitions that are possible within the prover, proving that each sequence is indeed possible, and then showing that the time of the sequence is less than the required time. Given the amount of reasoning required, there is ample opportunity to fall prey to the theorem prover pitfalls discussed in Section 1. Portions of these proofs can be automated with appropriate PVS strategies, which minimizes the number of opportunities the user has to waste and/or repeat work. For many untimed properties, the burden of theorem proving can be completely removed from the user. After performing the proofs of several systems using the encoding, PVS strategies were developed to assist the user in proving both of these types of properties.

These strategies were applied to a set of testbed systems. The specifications that comprised the testbed varied from the specification of a distributed mutual exclusion protocol to a phone switching system to a production facility. More specifically, the specifications include a number of standard benchmark systems: a bakery specification that describes the distributed mutual exclusion algorithm of [25], a cruise control system based on the description in [30], the elevator control system described in Section 2, a production cell specification based on the description in [17], a railroad crossing system based on the description in [26], and a stoplight specification adapted from the stoplight control system described in [11]. The testbed also includes the specification of an electronic scoring system for Olympic boxing based on a description of the system taken from the official 1996 Olympic web site [27]. Finally, the testbed includes a long distance telephony specification taken from [7]. The complete ASTRAL specifications of the testbed systems are available in [20].

As shown in Section 4.3, the ASTRAL proof obligations are inductive on the time domain, thus all have a base case. In the base case, each property must be shown to hold when the system is first initialized. The *try-base-case* strategy was developed to discharge these obligations. The try-base-case strategy introduces the initial_state axiom and then invokes the PVS *grind* command, which is a heavy-duty decision procedure that performs rewriting, skolemization, and automatic quantifier instantiation. The try-base-case strategy is sufficient in most cases to discharge the base case obligations automatically since many simplifications are possible at time zero. Table 1 lists the number of base case obligations in each testbed system and the number that were automatically proved using the try-base-case strategy.

There are three main cases in which this strategy fails. The first case is when imported variables are referenced in the property. In this case, it may be necessary to introduce information about the initial state of the other properties with the specification-dependent i_initial_state axiom. Then, the quantified formulas must be instantiated with the correct process types. The second case is when an immediate response is required such as "Call(trans1, now) → Start(trans1, now)". In this case, it is necessary for the user to prove that the required response will occur. This is almost always simpler in the base case than in the inductive cases, since at time zero all processes are idle and the state is completely known. Finally, try-base-case can fail when the base case

Table 1
Results of try-base-case on testbed system properties

System	Total base cases	Proved base cases
Bakery algorithm	2	2
Cruise control	2	2
Elevator	5	4
Olympic boxing	4	3
Phone	4	3
Production cell	9	8
Railroad crossing	3	1
Stoplight	1	0
Total	30	23

obligation contains complex definitions or quantifications that cannot be resolved by grind.

6.1. Untimed formulas

Properties of real-time systems do not consist solely of timing requirements. In fact, as shown in [23], more than half of the total number of properties in the testbed systems are of an untimed variety. Given this fact, it is crucial to provide assistance for both untimed as well as timed properties.

6.1.1. The try-untimed strategy

The *try-untimed* strategy was written to attempt the proofs of properties that do not involve time and only deal with combinations of state variables of a single process instance. For example, in the Elevator process type, one such property in the invariant section is "elevator_moving $\rightarrow \sim$door_moving". That is, whenever the elevator car is moving, the elevator door should not be in the process of opening or closing. This property was proved completely unassisted by the try-untimed strategy.

The basis of the try-untimed strategy is that in the interval T0 to T0$+\Delta$ of the proof obligations, the state variables either stay the same or one or more of them change. If the variables stay the same, then by the inductive hypothesis, the property holds at all times in the interval. If a variable changes during the interval, then by the semantics of ASTRAL, a transition ended at the time of the change. Furthermore, since transitions are nonoverlapping and, as discussed, Δ has been limited to a constant less than the duration of any transition, only a single transition end can occur within the interval. Fig. 5 depicts this situation. Let T1 be the time of such an end. Since no transition ended in the interval [T0, T1), the state variables must have stayed the same during that time period, thus the property holds by the inductive hypothesis. Similarly, since no transition ended in the interval (T1, T0 $+ \Delta$], the variables are unchanged in that region, thus the property holds in that region if it holds at T1. The bulk of the strategy is thus devoted to proving that the property holds at T1.

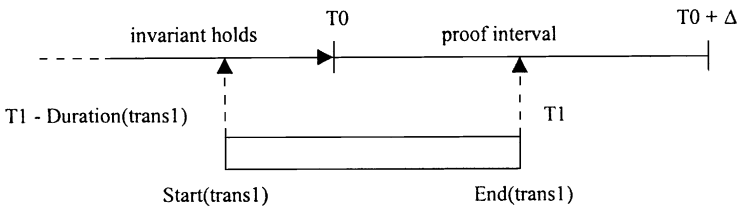

Fig. 5. Proof interval.

To prove this, it must be shown that all transition exit assertions preserve the property, thus the proof is split into a case for each transition and the transition's entry and exit clauses are asserted. Once again, since Δ was limited to less than the duration of any transition, the start of the transition occurred before $T0$, thus the property held at the start of the transition. From this point, a modified version of grind is invoked to finish the proof. The modified version, called *my-grind*, is shown below.

```
(defstep my-grind (&optional (if-match NIL))
    (then@
        (astral-expand-clause)
        (repeat (try (skosimp*) (assert) (skip)))
        (delete-bad)
        (grind :exclude("Start1" "Startn" "End1" "Endn" "Call1" "Calln"
                        "Change1" "Changen" "Issued_Call" "UQ" "Mod" "Div")
            :if-match if-match)))
```

The my-grind strategy is essentially grind with two optimizations. The main optimization consists of deleting expressions in the sequent that cannot be used effectively by PVS such as the timed operators. The supplementary strategy *delete-bad* uses string searches to find and delete formulas in the sequent that contain these "bad" expressions. When these definitions are expanded, grind attempts to automatically instantiate quantifiers in the expansion, which increases running time. Since grind cannot usually instantiate correctly in these situations, excluding the definitions saves significant time. My-grind works by first using another supplementary strategy, *astral-expand-clause*, to expand the ASTRAL definitions up to the clause level so that delete-bad will not miss expressions that are hidden in definitions. It then repeatedly tries the PVS commands *skosimp** (i.e. repeated skolemization and disjunctive simplification) and *assert* (i.e. a fast decision procedure that is the core of PVS) until no more simplifications can be made. This is done so that delete-bad will not delete terms that are separable from the "bad" terms. This results in a second optimization because in some cases, one of the repeated asserts will complete the proof without grind ever being invoked, which means the proof can be discharged very quickly. If assert does not complete the proof, delete-bad is executed followed by grind. Most of the operators that are removed by delete-bad are also excluded from rewriting by appropriate grind arguments. This is done so that definitions that are not expanded by astral-expand-clause, but that contain these operators, are not expanded by grind. Note that using grind's exclude option

Table 2
Results of try-untimed on testbed system properties

System	Applicable properties	Proved properties
Bakery algorithm	5	3
Cruise control	5	5
Elevator	2	2
Olympic boxing	1	1
Phone	17	10
Production cell	14	8
Railroad crossing	0	0
Stoplight	11	0
Total	55	29

without the expansion, simplification, and deletion of my-grind will not achieve the same performance gain because grind will still attempt to instantiate quantifiers that contain bad expressions, it will still split the proof when a bad expression is conjoined with another, and it will still attempt to use the unmodified bad expressions as instantiations for other quantified expressions.

6.1.2. Try-untimed results

Table 2 shows the results of using the try-untimed strategy to attempt the proofs of applicable invariant and schedule properties in the testbed systems. In this case, "applicable" means that the property is untimed and only references local state variables. The table shows that over half of the properties that are applicable were automatically discharged by the try-untimed strategy.

A side benefit of the try-untimed strategy is that even when it fails, it is still advantageous for the user to run because usually only very difficult cases will be left for the user to prove. When the strategy fails, it is due to one of three reasons. The first reason is that the user invoked the strategy on a timed property or one that involves imported variables. In this case, it is likely that most of the cases will fail, since try-untimed was not intended to deal with these types of properties. The second reason is that one or more transitions do not preserve the property. In this case, the user knows the exact transitions that failed since PVS will require further interaction to complete those cases. The user can correct the specification before continuing with other proofs. The last reason, which will be the most likely, is that it failed because there was not enough information in the entry assertion of a transition to prove the property. Usually, this occurs when the value of a variable in the formula to be proved is not explicitly stated in the entry assertion of the transition, but instead is implied by the sequences preceding that particular transition. For example, consider the elevator property "elevator_moving $\rightarrow \sim$ door_open". That is, the door must be closed while the elevator car is moving. After running the try-untimed strategy, all the transition cases are proved except for the "door_stop" case. The door_stop transition, shown below, stops the door in either the open or closed position after a suitable length of time from

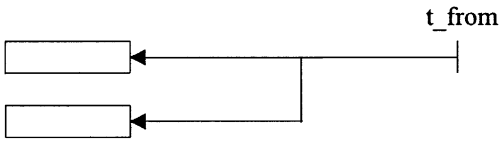

Fig. 6. An indeterminate backward step.

when the door started moving.

```
TRANSITION  door_stop
    ENTRY       [TIME:  door_stop_dur]
          door_moving
      &  now  -  t_move_door ⩾ Change(door_moving)
    EXIT
          ∼door_moving
      &  door_open = ∼ door_open′
```

The strategy fails for this case because it is possible for door_open to be set to true in the exit assertion and yet the value of elevator_moving is not stated in the entry assertion so can possibly be true if door_stop follows a transition in which elevator_moving is true. If elevator_moving is true and door_open is false when door_stop begins firing, then the formula will hold at the start of execution yet will not hold at the end of execution. In order to complete the proof of this property, it is necessary to consider the transitions that can fire immediately before door_stop. If the proof still cannot be completed, transitions must be considered further and further back in time. Eventually, the formula will be provable or a violation will occur.

6.1.3. The step-bw-indeterminate strategy

To assist the user in making such backward steps in untimed proofs, the *step-bw-indeterminate* strategy was developed. This strategy takes a time t_from and performs the necessary proof steps to derive the transitions that could have ended prior to this time as shown in Fig. 6. It is first shown that there is a transition that ended before t_from. The strategy attempts to discharge this subgoal by achieving a contradiction between the initial state and the state at t_from. This is possible because if no transition ends before t_from, then the variables could not have changed value since the initial state. The strategy then invokes my-grind, which in most cases will be sufficient to finish the proof. In the cases that it is not sufficient, the user must complete the proof by expanding timed operators or introducing relevant assumptions that require some transition to end between the initial state and t_from.

Since there is a transition that ended before t_from, there is a transition that ends last by an appropriate ASTRAL lemma. After it has been determined that some transition has ended last, the strategy then attempts to eliminate as many of the possible predecessors as possible by achieving a contradiction between the entry/exit of those

transitions and the state at t_from. This step is performed in a similar manner to proving the sequence generator obligations in the next section and fails for similar reasons. In this case, however, more information, such as the inductive invariant/schedule, is available to PVS, which makes this step more likely to succeed. When it fails, however, the user must prove the contradictions manually by expanding timed operators and/or stepping backward appropriately.

6.2. Transition sequence generator

Since sequencing is so important to proving some properties, it is useful to provide the user with a tool to view the transition sequences that can occur in a given process type. Such a tool can be used to estimate time delays between states, help the user visualize the operation of the system, assist in developing a plan of attack for a specific proof, and in some cases can be used to prove simple system properties. Unlike graphical state-machine languages in which the successor information is part of the specification, in textual languages such as ASTRAL, sequencing cannot be determined without more in-depth analysis. In addition, determining whether one transition is the successor of another in ASTRAL is undecidable since transition entry/exit assertions may be arbitrary first-order logic expressions. Many successors, however, can be eliminated based only on the simpler portions of the entry/exit assertions, such as boolean and enumerated variables. Based on this fact, a transition sequence generator tool has been developed.

6.2.1. Sequence generator proof obligations

The sequence generator first eliminates as many transition successors as possible. This is done by attempting the proof of an obligation trans1_not_trans2 for each pair of transitions (trans1, trans2) as shown below. Note that this obligation only states that some transition must end between trans1 and trans2 and does not exclude trans1 or trans2 from firing. The obligation is sufficient, however, to prove that a transition besides trans1 and trans2 must fire in between any firing of trans1 and trans2. If only trans1 and trans2 fire in between t1 and t2, then since $t2 - t1$ is finite and the durations of trans1 and trans2 are constant and nonnull, eventually a contradiction can be achieved by applying the theorem below repeatedly on an ever shortening interval. An obligation initial_not_trans1, as shown below, is also attempted to prove that each transition is not the first to fire after the initial state.

```
trans1_not_trans2: THEOREM
  (FORALL (t1, t2: time):
    t1 + Duration(trans1) ⩽ t2 AND
    Fired(trans1, t1) AND
    Fired(trans2, t2) IMPLIES
    (EXISTS (trans3: transition, t3: time):
      t1+Duration(trans1) <
        t3+Duration(trans3) AND
      t3+Duration(trans3) ⩽ t2 AND
      Fired(trans3, t3))))
```

```
initial_not_trans1: THEOREM
  (FORALL (t1: time):
    Fired(trans1, t1) IMPLIES
    (EXISTS (trans2: transition, t2: time):
      t2+Duration(trans2) ⩽ t1 AND
      Fired(trans2, t2)))
```

Table 3
Transition successors of testbed systems

System	Process type	Maximum successors	Actual successors	Computed successors
Bakery algorithm	Proc	42	8	25
Cruise control	Accelerometer	2	2	2
	Speed_Control	132	76	94
	Speedometer	2	2	2
	Tire_Sensor	2	2	2
Elevator	Elevator	42	13	24
	Elevator_Button_Panel	6	4	4
	Floor_Button_Panel	20	14	14
Olympic boxing	Judge	2	2	2
	Tabulate	12	4	6
	Timer	6	3	3
Phone	Central_Control	420	235	312
	Phone	110	50	69
Production cell	P_Crane	156	13	36
	P_Deposit	6	3	3
	P_Deposit_Sensor	6	3	3
	P_Feed	20	14	14
	P_Feed_Sensor	6	3	3
	P_Press	42	7	7
	P_Robot	420	21	129
	P_Table	72	9	21
Railroad crossing	Gate	20	7	7
	Sensor	6	3	3
Stoplight	Controller	506	92	198
	Sensor	6	3	3
Total		2064	593	986

6.2.2. Sequence generator strategies

The PVS strategies *try-seq-gen* and *try-seq-gen-0* were written to automatically discharge these obligations. The try-seq-gen strategy uses abstract machine axioms to introduce the entry and exit assertions of trans1, the entry assertion of trans2, and the fact that if nothing ended between the end of trans1 and the start of trans2, then all variable values remained constant during this time. Once all of this information is present, the strategy invokes my-grind. The try-seq-gen-0 strategy is similar but uses the initial clause of the process in place of the information about trans1.

6.2.3. Sequence generator results

Table 3 shows the results of using these strategies to compute the successors for each process type of the set of testbed systems. For each process type, the table shows the maximum number of successors, the number of successors that are provably possible, and the number that were computed automatically using the try-seq-gen strategies.

There are two main factors that contribute to the difference between the number of successors that are provably possible and the number computed by the try-seq-gen

strategies in the testbed systems. The first factor is that entry assertions do not usually constrain all of the state variables of a process. For example, the entry assertion of the door_stop transition, shown in Section 6.1, constrains the value of door_moving, but does not constrain the value of elevator_moving.

When proving that the arrive transition, shown below, cannot follow door_stop, PVS does not have information about the value of elevator_moving at the start of door_stop, which is only derivable from the transitions preceding door_stop. Thus, PVS must assume an arbitrary symbolic value for elevator_moving. It is possible that elevator_moving is true, thus PVS cannot eliminate the possibility that arrive immediately follows door_stop. It is provable that this is not the case, however, because it is not possible to find a sequence of transitions starting from the initial state in which arrive can immediately follow door_stop. The only possible predecessors to door_stop are open_door and close_door. Open_door sets elevator_moving to false in its exit assertion, thus if open_door immediately precedes door_stop, arrive cannot follow door_stop. Similarly, it is possible to show that close_door must be preceded by door_stop, which is preceded by open_door. Thus, arrive cannot follow door_stop.

```
TRANSITION  arrive
ENTRY       [TIME:  arrive_dur]
        elevator_moving
    &  FORALL  t:  time
            (   t ⩽ now
            &  (    End(move_down,  t)
                |    End(move_up,  t))
            ⟶       now  -  t_move ⩾ t)
    &  FORALL  t,  t1:  time
            (   t ⩽ now
            &  End(arrive,  t)
            &  (    End(move_up,  t1)
                |    End(move_down,  t1))
            ⟶       t < t1)
EXIT
        IF  going_up′
        THEN  position = position′  +  1
        ELSE  position = position′  -  1
        FI
```

In order to improve the accuracy of the sequence generator for these processes, it would be necessary to examine sequences back to a transition that causes a contradiction. This is a nonterminating procedure, however, whenever the second transition of a successor obligation actually is a successor of the first, thus it is necessary to specify termination conditions such as a specific number of transitions into the past or similar criteria. In general, this procedure is not worth the additional time it would require unless the number of successors that could be eliminated using a small number of backward steps is significantly higher than the number of actual successors.

As an alternative, the user can fully constrain all of the state variables in the entry assertions.

The second factor that contributes to the difference between the number of provable successors and the number computed by the try-seq-gen strategies is the use of timed operators to define the sequencing between different operations. For example, the end operator is used in the arrive transition to prevent arrive from following itself. In the proof of the successor obligation arrive_not_arrive, arrive fires at t1 and t2 and no other transition fires in between. By the last conjunct of arrive's entry assertion, there must be an end to move_up or move_down between the last time arrive ended (t1+arrive_dur) and the next time it fires (t2), which contradicts the fact that no transition fires in between t1 and t2. This proof cannot be carried out without the use of the end operator. The definition of the end operator within PVS, however, is quite complex with several quantifiers, thus there is little hope that PVS could automatically prove such an obligation. For this reason, my-grind is applied, which prevents work from being wasted.

6.2.4. Parameterized transition sequences

When a transition is parameterized, such as the request_floor transition of the Elevator_Button_Panel process shown in Section 2, each set of parameters represents one possible choice that a process can make. Usually, the start of a transition with one set of parameters does not preclude the start of the same transition with a different set of parameters immediately afterward. Thus, the sequences generated for parameterized transitions do not usually give any helpful information to the user since essentially any transition can follow any other.

Since the standard sequence generator proof obligations do not ordinarily produce a useful result for parameterized transitions, a parameterized extension has been added to the sequence generator. In this extension, if two transitions have the same parameter list (i.e. the same number of parameters and parameter types), the successor proof obligations are attempted assuming that the parameters are the same. That is, the sequences are generated with a fixed set of parameters among consecutive transitions. This is useful for finding the sequence of transitions in a single "thread". For example, by keeping the parameters fixed in the Elevator_Button_Panel, it can be determined that the same floor cannot be requested twice in a row. The numbers in Table 3 were computed using the parameterized extension. The numbers for the Elevator_Button_Panel, Central_Control, and Controller processes are the only processes affected by this extension.

6.2.5. Transition sequence construction

After the successors have been computed, the sequence generator constructs transition sequences based on input from the user, which includes the first and last transitions, the direction to generate sequences from the first transition, the maximum number of transitions per sequence, and the maximum number of sequences. There is also an option to disallow sequences in which the same transition appears more than once (besides as the

first or last transition). The user must provide the maximum number of transitions per sequence and if the search is backward, must provide the first transition. The sequence generation process is completely automatic and is available as a component of the ASTRAL Software Development Environment (SDE) [22]. The ASTRAL SDE constructs the sequence generator obligations, invokes PVS, runs the proof scripts, retrieves the results, and then generates the sequences according to the user query. Since running the proof scripts can be time consuming, the results are saved between changes to the specification, so that sequences from previous proof attempts can be quickly displayed.

For each sequence generated, an approximate running time of the sequence is constructed by analyzing the entry assertion of each transition. Entry assertions depend on the values of local and imported variables, the call/start/end times of local and imported transitions, and the current time in the system. Transitions that only depend on local variables and/or the start/end times of local transitions will always fire immediately after another transition. Transitions that reference the current time, however, may be delayed some amount of time before firing. For example, the door_stop transition, shown in Section 6.1, fires at least t_move_door after the door starts moving. Similarly, transitions may wait indefinitely for a change to an imported variable, a call/start/end to an imported transition, or a call to a local transition from the external environment. The three types of delays are denoted *delay_T* for a time delay, *delay_O* for a delay because of the other processes in the system, and *delay_E* for a delay due to the external environment.

The sequence generator is complete (i.e. if a sequence is possible it will appear as a result) without the parameterized extension since the successor obligations are performed using the PVS encoding, which will only eliminate a successor if it is derivable that it cannot occur. The sequence generator is not complete with the parameterized extension because it does not display any sequences in which two parameterized transitions with the same parameter lists are given different parameters. In this case, utility was chosen over completeness.

The accuracy of the sequence generator can be improved by manually performing the proofs of those successor obligations that actually can be proved but could not be automatically proved by the try-seq-gen strategies. The time used to run the proof scripts or to refine the performance of the sequence generator is not wasted because any successor eliminated can be used as a lemma in the main proof obligations.

As a simple example of a sequence generator query, consider the door_stop case that failed in the try-untimed proof of "elevator_moving $\rightarrow \sim$door_open" in Section 6.1. The user may wish to view the predecessors to door_stop to see if the proof can be completed quickly or if a violation is possible involving the door_stop transition. Fig. 7 shows the sequence generator dialog box and the second of the three sequences generated from the query.

Three sequences are returned to the user, which show three possible predecessors to door_stop: close_door, open_door, and arrive. If close_door fires before door_stop, the door is closed when door_stop completes firing, thus the property trivially holds. The open_door transition sets elevator_moving to false, thus the property also trivially holds

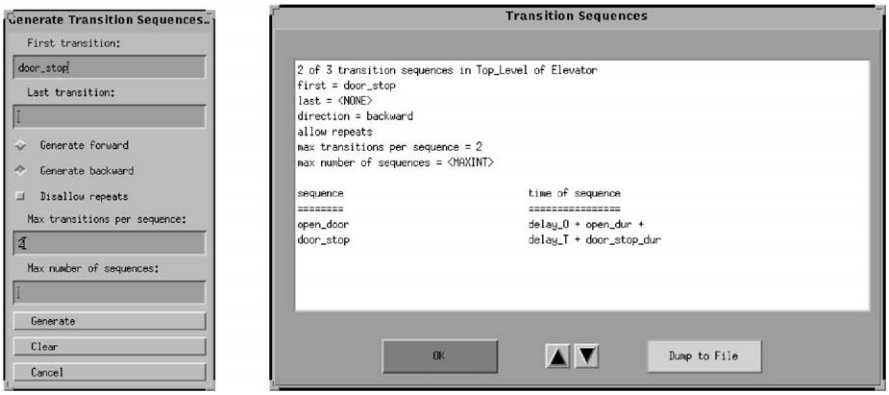

Fig. 7. Sequence generator dialog box and query result.

if open_door fires before door_stop. The arrive transition, shown earlier, requires the elevator car to be moving to fire. By the inductive hypothesis, the door is closed when it fires, thus if arrive precedes door_stop, the invariant can be violated because the elevator car is moving and door_stop sets door_open to true. Therefore, the user knows that to complete the proof, it must be shown that arrive cannot fire immediately before door_stop. The arrive case is another example of a successor that the sequence generator could not eliminate automatically and yet is not actually possible after further analysis. Thus, the user must consider the predecessors of arrive and continue the proof process in a similar manner until the property is proved. Additional uses of the transition sequence generator can be found in [21].

6.3. Timed formulas

The proof assistance for timed formulas cannot be fully automated like many untimed formulas, thus the strategies for timed formulas are based on finding the transition sequences that are possible in a given process type. All ASTRAL requirements are based on the current time, the values of local variables, the call, start, and end times of local transitions, the values of imported variables, and the call, start, and end times of imported transitions. From a sequence of transitions, all of this information can be derived, thus any ASTRAL requirement can be proven (if possible) by analyzing transition sequences. The start and end times of local transitions can be found directly from the sequence. The values of local variables can be derived from the entry and exit assertions of each transition in the sequence. The values of imported variables and the call, start, and end times of imported transitions can be derived from the imported variable clause using the values of exported local variables and the start and end times of exported local transitions. The call times of exported local transitions can be derived from the environment clause using the values of exported local and imported variables and the start and end times of exported local and imported transitions. Finally, a symbolic value for the current time can be derived from the sequence using the other information and the entry assertion of each transition.

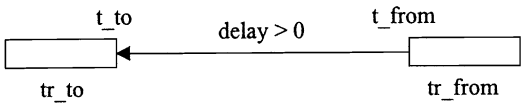

Fig. 8. A delayed backward step.

Timed transition steps are broken down into forward and backward steps and immediate and delayed steps and are always computed from a given time that a specific transition fired. *Forward steps* compute the transitions that can fire right after the given transition while *backward steps* compute the transitions that could have fired right before the given transition. *Immediate steps* compute the transitions that can fire immediately after or before the given transition while *delayed steps* compute the transitions that can fire after or before the given transition with some specific delay. PVS strategies have been developed for all of these types of steps in the *step-fw-immediate*, *step-fw-delay*, *step-bw-immediate*, and *step-bw-delay* strategies. These strategies are repeatedly applied until enough information is present in the sequent to finish the proof. Only the step-bw-delay strategy will be discussed, which is the most involved of all of the strategies. This strategy is highly complex, requiring 207 lines of strategy code for its definition. An equivalent proof by hand would contain numerous opportunities to waste time and duplicate work as discussed in Section 1. By encapsulating all of the steps within a predefined strategy and utilizing the most efficient ordering in a highly automated procedure, the user's potential efficiency and effectiveness is maximized.

The *step-bw-delay* strategy computes the transitions that could have fired a given time right before a given transition. This strategy takes a "source" transition, tr_from, the time it fired, t_from, a "destination" transition, tr_to, and the time it is to end, t_to. It then performs the necessary proof steps to show that tr_to is the last transition to end and that it ends at t_to as shown in Fig. 8. In order to show this, five subgoals must be proved.

- $t\_from - t\_to > 0$: This strategy is only meant to be used when there is a delay between the start of tr_from and the end of tr_to. If there is no delay, then the step-bw-immediate strategy should be used instead since more of it can be fully automated. The strategy attempts to discharge this subgoal with the PVS assert command. This may or may not finish the proof depending on the forms of t_from and t_to. If either of these expressions has a complex form, it may be necessary for the user to complete this proof by introducing type predicates for the terms in each expression.
- *Some transition ended before* t_from: The strategy attempts to discharge this subgoal by achieving a contradiction between the initial state and the state at t_from. This is possible because if no transition ends before t_from, then the variables could not have changed value since the initial state. The strategy invokes my-grind, which in most cases will be sufficient to finish the proof. In the cases where it is not sufficient, the user must complete the proof by expanding timed operators or introducing relevant assumptions that require some transition to end between the initial state and t_from.

- *The transition that ended last was tr_to*: Since there is a transition that ended before t_from, there is a transition that ends last by an appropriate ASTRAL lemma. The strategy attempts to discharge this subgoal by showing that the transitions besides tr_to could not have been the last to end or else a contradiction could be achieved between the entry/exit of those transitions and the entry of tr_from. This step is performed in a similar manner to proving the sequence generator obligations and fails for similar reasons. In this case, however, more information, such as the inductive invariant/schedule, is available to PVS, which makes this step more likely to succeed. When it fails, however, the user must prove the contradictions manually by expanding timed operators and/or stepping backwards appropriately.
- *tr_to did not end before t_to*: Once it is shown that tr_to was the last transition to end, it must be shown that tr_to ended at t_to. The strategy attempts to show that if tr_to ended earlier than t_to, then tr_from would fire earlier than t_from. In this case, if tr_from ended before t_from, then a contradiction is achieved with the fact that nothing ended between the end of tr_to and t_from. If tr_from did not end before t_from, then by trans_mutex, tr_from could not fire at t_from. The main thing the user must prove in this step is that tr_from is enabled after the given delay elapses from the end of tr_to. This will usually require expanding timed operators in the entry assertion of tr_from.
- *tr_to did not end after t_to*: The strategy attempts to discharge this subgoal in a similar manner to the previous step. In this case, it must be shown that if tr_to ends later than t_to, then tr_from could not be enabled (hence fire) at t_from. Since the entry assertion of tr_from is most likely dependent on timed operators, the user must expand these operators appropriately.

  If tr_from is not delayed due to timed operators, then it must be delayed by other processes or the external environment. In these cases, it must be shown that the change to the operating environment was delayed in response to some change made by tr_to. Otherwise, it will not be possible to prove this subgoal because the operating environment must have changed at t_from, which means that tr_from will still be enabled.

These five subgoals are sufficient to show that tr_to ends at t_to. The first subgoal shows that there is a nonzero delay between the end of tr_to and the start of tr_from. The second subgoal shows that there has been some transition that has fired in the execution history of the process. If no transition has fired, then tr_to cannot possibly have fired before tr_from. The third subgoal shows that the last transition to end was tr_to. Finally, the last two subgoals show that tr_to did not fire too early or too late, respectively.

6.4. Theorem proving results

Overall, 21 strategies were developed to assist in the analysis of ASTRAL specifications. Table 4 shows the results of using PVS and the developed strategies to prove the proof obligations of the testbed systems. The results of the theorem prover proofs and

Table 4
Results of theorem proving on testbed systems

System	Total obligations	Attempted obligations	Completed obligations	Prover commands
Bakery algorithm	21	18	17	466
Cruise control	9	9	9	535
Elevator	33	13	12	234
Olympic boxing	18	17	17	1073
Phone	51	25	16	172
Production cell	69	37	31	903
Railroad crossing	14	10	8	1367
Stoplight	24	18	0	29
Total	239	147	110	4779

earlier proofs by hand are the basis for a systematic analysis methodology described in [21] in which tool-supported guidance is given for constructing proof sketches by hand, which are then carried out in a similar fashion within PVS. As can be seen, approximately half of the total number of proof obligations were completely discharged using the prover. All of the obligations of the cruise control system and the Olympic boxing scoring system specifications were completely discharged with the exception of the global schedule of the scoring system. This schedule, however, is not provable due to a flaw in the scoring system itself and not in the specification. Namely, it is possible for a boxer to obtain more total points and yet still lose the fight.

In Table 4, the number of proof obligations attempted indicates how many proofs were started, but not completed. The meaning of this number varies from system to system. In some cases, such as the railroad crossing, a significant portion of the proofs that were not completed were performed. For example, in the proof of the Gate schedule, one of the two worst cases was proved, which demonstrated how all of the others could be proved. In the case of the obligations of the stoplight control system, however, only a small number of approaches were tried. The number of prover commands gives an estimate of the effort associated with each system. These numbers only include the latest attempt of each obligation and do not include earlier attempts or backtracking, which would make the numbers significantly higher.

The systems besides the cruise control system and the scoring system were not completely proved due to a number of factors. The foremost reason is that as more and more of the obligations were discharged, it became evident that most of the proofs had similar themes and could be proved using the same techniques as earlier proofs. Thus, once enough mechanisms were developed to deal with the most common themes, it became less critical to actually complete every proof. This was the case for the bakery algorithm, the production cell, and the railroad crossing specifications. The other factor is that some of the processes exhibit behavior that is extremely nontrivial to reason about within a theorem prover. This was the case for the elevator control system and phone system.

One of the central themes of the proofs of the response requirements of the elevator is finding the maximum number of full iterations that the elevator can execute before

the requested floor is reached. The main difficulty arises when it must be proved that this number is actually the worst case and that the other cases are subsumed. This type of proof is common to all real-time systems, but in the Elevator process, it is significantly more difficult to prove. Unlike other process types in which the worst case usually covers one or two basic process cycles, in the Elevator process, the worst case can encompass an arbitrary number of cycles. This is because the Elevator stores an iteration count (i.e. the current position) that affects its behavior and that is used in the requirements. The maximum position is a symbolic value and at each of the arbitrarily many floors, a series of complex actions must be performed. The worst case must then be shown to be the appropriate case where the elevator travels the maximum symbolically constrained number of floors with complex reasoning required at each floor. Processes with this type of behavior are discussed in [23] and are referred to as *iterative single-threaded processes*. That is, they record some notion of which iteration they are on such as a loop count or queue length and must perform a complex series of actions in each iteration. In order to deal with these types of processes, new theorem prover techniques must be developed.

One of the central themes of the proofs of response requirements of the phone system is finding the maximum number of phones that can have requests outstanding at any given time. This is equivalent to finding the cardinality of the set of transitions that service those requests that are enabled at a given time. In a hand proof, such a cardinality can be found fairly quickly based mostly on human ingenuity and "hand waving". In PVS, however, cardinality proofs are extremely complex and become even more so when the set predicate is nontrivial. For the set of enabled transitions, the set predicate (i.e. is a specific transition enabled at the given time) is highly nontrivial as it depends on the arbitrary first-order logic expressions of the transition entry assertion as well as the execution history of the process and the behavior of the operating environment. Thus, determining the cardinality of this set within PVS becomes intractable. Further research is necessary to make such a proof feasible.

Although additional theorem prover techniques are needed for process types such as the elevator control system and the phone system, these processes compromise only two of 25 of the processes in the testbed systems. Given that the testbed systems are a random sample taken from existing literature, it is likely that simpler process types make up the significant majority of real-world systems as well. This is also a reasonable assumption because every complex process is inevitably surrounded by a number of simple processes such as buttons, sensors, and other input/output processes that support it. This means that the techniques described in this paper should be directly applicable to most real-time systems.

7. Related work

The encoding of several hardware description languages into HOL is discussed in [4]. Two different encoding styles are introduced, referred to as *deep encodings* and

shallow encodings. In a shallow encoding, the theorem prover representation mirrors the syntactic representation of the language being encoded. In a deep encoding, however, the semantics of the language is encoded within the logic of the prover without regard to syntactical features. In this terminology, the ASTRAL encoding is a shallow encoding since it was desired to keep the PVS representation as close as possible to the ASTRAL language.

The real-time temporal logic TRIO has been encoded into PVS [1] as discussed in Section 4.1. TRIO is very closely related to ASTRAL and is, in fact, the basis for the core ASTRAL logic [13]. TRIO does not have the structural mechanisms or abstract machine definition of ASTRAL, however, thus is a lower-level formalism. The encoding style used for TRIO differs from that of ASTRAL as discussed in Section 4.1, but the goal of keeping the encoding similar to that of the base language is the same. There is no discussion, however, of any strategies developed that can assist the user in performing proofs.

Conversely, the Duration Calculus encoding [28] is very similar to ASTRALs as discussed in Section 4.1 and is also supported by a large number of automated strategies. The duration calculus is a real-time temporal logic, however, as opposed to the state machine approach of ASTRAL.

Several real-time state machine languages have also been encoded into theorem provers. The Timed Automaton Model has been encoded into PVS [2] and Timed Transition Systems into HOL [16]. These languages are based on interleaved concurrency, however, which makes their semantics simpler than those of ASTRAL. Additionally, timed transition systems are not defined in terms of arbitrary first-order logic expressions and do not have the complex subtyping mechanisms that are available in ASTRAL.

The timed automaton model encoding is supported by PVS strategies similar to the ones found in Section 6 [3]. Several of the strategies correspond closely with the strategies developed for ASTRAL. Most notably, the last- and first-event strategies have a function similar to the step-bw and step-fw strategies. This indicates that such strategies are useful for many different real-time specification languages and not just ASTRAL. Although [3] does provide several useful techniques for allowing the PVS proofs to correspond closely to hand proofs, what is lacking is any guidance on how the hand proof is to be constructed as is discussed for ASTRAL in [21].

An encoding of ASTRAL into PVS was reported in [5, 6], but this encoding is based on a definition of ASTRAL that has been developed independently at Delft University based on earlier ASTRAL work in [12, 13]. The ASTRAL definition in [12, 13] did not include the notion of an external environment, thus did not include the call operator, environmental assumptions, or schedules. The Delft definition has diverged from the work reported in [7] and [8] and has essentially become a different language. It includes only a small subset of the full set of ASTRAL operators and typing options, does not include all of the sections of an ASTRAL specification, and defines only a small fraction of the axiomatization of the ASTRAL abstract machine. In addition, it is

based on a discrete time domain and proofs are performed with a global view of the system rather than using a modular approach.

8. Conclusions and future work

This paper has discussed the adaptation of the PVS theorem prover for performing analysis of real-time systems written in the ASTRAL formal specification language. The encoding attempts to minimize the differences between an ASTRAL specification and its PVS equivalent to allow the user to interpret results more easily. The decisions made for ASTRAL on a number of encoding issues were highlighted. From the proof attempts of a variety of different real-time systems, a number of strategies were developed that encapsulate frequently occurring proof patterns and provide significant assistance to the user during PVS proofs. Finally, a transition sequence generation tool was implemented using the PVS encoding that provides valuable information to the user throughout the proof process.

A number of issues still need to be addressed in future work. The implementation clause of ASTRAL, which is used to map relationships between upper and lower level specifications, needs to be incorporated into the translator, as well as the inter-level proof obligations used to show that an implementation is consistent with the level above. The refinement mechanism described in [9] has recently been completely reworked in [24], thus the translation had been postponed until this new mechanism was put in place.

A number of enhancements to the sequence generator can be added. For instance, it is useful to provide a more powerful interface. For example, a query interface could be added to answer queries such as whether a given transition can ever occur between two other specified transitions. It is also possible to construct a symbolic expression for the values of the state variables at the end of each sequence by examining the entry and exit assertions of each transition.

In general, more proofs need to be performed for different ASTRAL systems using their PVS translations. In studying the proofs performed for many systems, more proof patterns may be discovered that can be incorporated into suitable PVS strategies. The patterns may also lead to the definition of useful lemmas that can be proven in advance and added to the ASTRAL–PVS library for future use. It is also worthwhile to investigate whether the structure of the ASTRAL specification determines which lemmas and strategies are most applicable to a given formula type.

Finally, as discussed in Section 6.4, there is a need for additional theorem prover techniques for process types similar to the elevator control system and phone system. In [21], it is discussed how to statically identify these types of processes. For process types similar to the elevator system, it is necessary to support “worst case” reasoning over iteration counts and to allow the other cases to be implicitly subsumed. For

process types similar to the phone system, it is necessary to support reasoning about the cardinality of complex sets. To provide the necessary support for these processes types, a more in-depth study of the capabilities of the theorem prover is needed.

Appendix

```
SPECIFICATION Elevator_System
  GLOBAL SPECIFICATION Elevator_System
    PROCESSES
              the_elevator: Elevator,
              the_elevator_buttons: Elevator_Button_Panel,
              the_floor_buttons: array [ 1..n_floors ]  of Floor_Button_Panel
    TYPE
              pos_integer: TYPEDEF i: integer ( i > 0 ) ,
              pos_real: TYPEDEF r: real ( r > 0 ) ,
              floor: TYPEDEF i: pos_integer ( i <= n_floors )
    CONSTANT
              n_floors: pos_integer,
              request_dur, clear_dur: pos_real,
              t_service_request, t_move, t_stop, t_move_door: pos_real
    AXIOM
      /* clear_request must be able to fire no matter how many requests are made
         while the elevator door is opening */
              ( clear_dur + n_floors * request_dur < t_move_door )
      /* must be at least 2 floors in the building */
            &  ( n_floors >= 2 )
    SCHEDULE
      /* any request must be serviced within time t_service_request */
              FORALL f: floor
                       ( the_elevator_buttons.Call ( request_floor ( f ) ,
                           now - t_service_request )
                     ->   EXISTS t: time
                                  ( now - t_service_request < t
                                  & t <= now
                                  & past ( the_elevator.position, t )  = f
                                  & past ( Change ( the_elevator.door_open, t ) , t )
                                  & past ( the_elevator.door_open, t )  )  )
            & FORALL f: floor
                       ( f ~= n_floors
                       & the_floor_buttons [ f ] .Call ( request_up,
                           now - t_service_request )
                     ->   EXISTS t: time
                                  ( now - t_service_request < t
                                  & t <= now
                                  & past ( the_elevator.position, t )  = f
                                  & past ( Change ( the_elevator.door_open, t ) , t )
                                  & past ( the_elevator.door_open, t )
                                  & past ( the_elevator.going_up, t )  )  )
            & FORALL f: floor
                       ( f ~= 1
                       & the_floor_buttons [ f ] .Call ( request_down,
                           now - t_service_request )
                     ->   EXISTS t: time
                                  ( now - t_service_request < t
                                  & t <= now
                                  & past ( the_elevator.position, t )  = f
                                  & past ( Change ( the_elevator.door_open, t ) , t )
                                  & past ( the_elevator.door_open, t )
                                  & ~past ( the_elevator.going_up, t )  )  )
```

```
END Elevator_System
PROCESS SPECIFICATION Elevator
  LEVEL Top_Level
    IMPORT
              pos_real, floor, request_dur, the_elevator_buttons, the_floor_buttons,
              the_elevator_buttons.floor_requested, the_elevator_buttons.request_floor,
              the_floor_buttons.up_requested, the_floor_buttons.down_requested,
              the_floor_buttons.request_up, the_floor_buttons.request_down, t_stop,
              t_move, t_move_door, t_service_request, n_floors
    EXPORT
              position, going_up, door_open, moving, door_moving
    CONSTANT
              move_dur, arrive_dur, open_dur, close_dur, door_stop_dur: pos_real
    VARIABLE
              position: floor,
              going_up, door_open, moving, door_moving: boolean
    AXIOM
      /* t_service_request must be big enough to handle the worst case. One instance of
         the worst case is when the elevator is moving up from floor 1 to 2 and 2 has
         not been requested on the elevator panel nor has any request been made on 2's
         button panel. Let t_arrive be the next time such that End(arrive, t_arrive).
         up_request and down_request are simultaneously called on floor 2 an "instant"
         after t_arrive - 2 * request_dur and down_request fires first. In addition,
         every floor in the building (besides 2) has up_requested (except the top floor)
         and down_requested (except the bottom floor). Thus, the up request is not
         posted in time for the elevator to service it and the elevator must stop and
         open the door at every floor up to the top, back down to the bottom, and back
         up to 2. */
               ( t_service_request >= 2 * request_dur + move_dur + t_move + arrive_dur +
                 ( 2 * n_floors - 3 )  *
                  ( open_dur + t_move_door + door_stop_dur + t_stop + close_dur +
                      t_move_door + door_stop_dur + request_dur + move_dur + t_move +
                      arrive_dur )  + open_dur + t_move_door + door_stop_dur )
    DEFINE
              request_above ( f0: floor ) : boolean ==
                      EXISTS f: floor
                               ( f > f0
                               &  ( the_elevator_buttons.floor_requested ( f )
                                 |  the_floor_buttons [ f ] .up_requested
                                 |  the_floor_buttons [ f ] .down_requested )  ) ,
              request_below ( f0: floor ) : boolean ==
                      EXISTS f: floor
                               ( f < f0
                               &  ( the_elevator_buttons.floor_requested ( f )
                                 |  the_floor_buttons [ f ] .up_requested
                                 |  the_floor_buttons [ f ] .down_requested )  )
    INITIAL
              position = 1
            & going_up
            & ~door_open
            & ~moving
            & ~door_moving
    INVARIANT
      /* the elevator door must stay closed while the elevator is moving */
               ( moving
            ->   ~door_open
               & ~door_moving )
    CONSTRAINT
      /* if the elevator changes direction, there cannot be an outstanding request in
           the old direction */
               ( going_up
```

```
          & ~going_up'
       ->   ~request_below' ( position' )  )
       &  ( ~going_up
          & going_up'
       ->   ~request_above' ( position' )  )
SCHEDULE
  /* if the elevator is moving in some direction, there must be an outstanding
       request in that direction */
          ( moving
          & going_up
       ->   request_above ( position )  )
       &  ( moving
          & ~going_up
       ->   request_below ( position )  )
  /* any request must be serviced within time t_service_request */
       &  ( FORALL f: floor
                  ( the_elevator_buttons.Call ( request_floor ( f ) ,
                      now - t_service_request )
                ->   EXISTS t: time
                           ( now - t_service_request < t
                           & t <= now
                           & past ( position, t )  = f
                           & past ( Change ( door_open, t ) , t )
                           & past ( door_open, t )  )  )  )
       &  ( FORALL f: floor
                  ( f ~= n_floors
                  & the_floor_buttons [ f ] .Call ( request_up,
                      now - t_service_request )
                ->   EXISTS t: time
                           ( now - t_service_request < t
                           & t <= now
                           & past ( position, t )  = f
                           & past ( Change ( door_open, t ) , t )
                           & past ( door_open, t )
                           & past ( going_up, t )  )  )
          & FORALL f: floor
                  ( f ~= 1
                  & the_floor_buttons [ f ] .Call ( request_down,
                      now - t_service_request )
                ->   EXISTS t: time
                           ( now - t_service_request < t
                           & t <= now
                           & past ( position, t )  = f
                           & past ( Change ( door_open, t ) , t )
                           & past ( door_open, t )
                           & ~past ( going_up, t )  )  )  )
IMPORTED VARIABLE
  /* buttons only clear after elevator has arrived and started opening the doors */
          ( FORALL f: floor
                  ( Change ( the_elevator_buttons.floor_requested ( f ) , now )
                  & ~the_elevator_buttons.floor_requested ( f )
                ->   EXISTS t: time
                           ( Change [ 2 ]
                               ( the_elevator_buttons.floor_requested(f)) < t
                           & t <= now
                           & past ( position, t )  = f
                           & ~past ( door_open, t )
                           & past ( door_moving, t )  )  )  )
       &  ( FORALL f: floor
                  ( f ~= n_floors
                  & Change ( the_floor_buttons [ f ] .up_requested, now )
```

```
                    & ~the_floor_buttons [ f ] .up_requested
                ->   EXISTS t: time
                           ( Change [ 2 ]
                               ( the_floor_buttons [ f ] .up_requested )  < t
                           & t <= now
                           & past ( position, t )  = f
                           & ~past ( door_open, t )
                           & past ( door_moving, t )
                           & past ( going_up, t )  )  )  )
       &  ( FORALL f: floor
                ( f ~= 1
                & Change ( the_floor_buttons [ f ] .down_requested, now )
                & ~the_floor_buttons [ f ] .down_requested
                ->   EXISTS t: time
                           ( Change [ 2 ]
                               ( the_floor_buttons [ f ] .down_requested ) < t
                           & t <= now
                           & past ( position, t )  = f
                           & ~past ( door_open, t )
                           & past ( door_moving, t )
                           & ~past ( going_up, t )  )  )  )
  /* the top floor never has an up request and the bottom floor never has a down
       request */
       &  ( ~the_floor_buttons [ n_floors ] .up_requested )
       &  ( ~the_floor_buttons [ 1 ] .down_requested )
  /* requests cannot be made of the elevator to stop at a floor between when the
       door starts opening on that floor until when it starts closing */
       &  ( Change ( door_moving, now )
          & door_moving
          & door_open
       ->   FORALL t: time
                ( t >= Change [ 2 ]  ( door_moving )
                ->   ~the_elevator_buttons.Call ( request_floor ( position ), t))
  /* requests cannot be made of the elevator to stop at a floor between when the
       door starts opening on that floor until when it starts closing */
       &  ( Change ( door_moving, now )
          & door_moving
          & door_open
       ->   FORALL t: time
                ( t >= Change [ 2 ]  ( door_moving )
                ->    ( past ( going_up, t )
                   ->   ~the_floor_buttons [ position ] .Call ( request_up, t ) )
                   &  ( past ( ~going_up, t )
                   ->   ~the_floor_buttons[ position ].Call ( request_down, t))))
TRANSITION move_up
  ENTRY           [ TIME : move_dur ]
            ~door_open
          & ~door_moving
          & request_above ( position )
          &  ( going_up
             |  ~going_up
             & ~request_below ( position )
             & ~the_floor_buttons [ position ] .up_requested )
          &  ( End ( arrive, now )
             & ~the_elevator_buttons.floor_requested ( position )
             & ~the_floor_buttons [ position ] .up_requested
             |  FORALL t, t1: time
                        ( Change ( moving, t )
                        & Change ( door_open, t1 )
                     ->   t < t1
                        & now >= t1 + request_dur )  )
```

```
  EXIT
            moving
          & going_up
TRANSITION move_down
  ENTRY           [ TIME : move_dur ]
            ~door_open
          & ~door_moving
          & request_below ( position )
          &  ( ~going_up
            |  going_up
             & ~request_above ( position )
             & ~the_floor_buttons [ position ] .down_requested )
          &  ( End ( arrive, now )
             & ~the_elevator_buttons.floor_requested ( position )
             & ~the_floor_buttons [ position ] .down_requested
            |  FORALL t, t1: time
                        ( Change ( moving, t )
                        & Change ( door_open, t1 )
                     ->   t < t1
                        & now >= t1 + request_dur )  )
  EXIT
            moving
          & ~going_up
TRANSITION arrive
  ENTRY           [ TIME : arrive_dur ]
            moving
          & FORALL t: time
                     ( t <= now
                     &  ( End ( move_down, t )
                       |  End ( move_up, t )  )
                   ->   now - t_move >= t )
          & FORALL t, t1: time
                     ( t <= now
                     & End ( arrive, t )
                     &  ( End ( move_up, t1 )
                       |  End ( move_down, t1 )  )
                   ->   t < t1 )
  EXIT
            IF
                    going_up'
            THEN
                    position = position' + 1
            ELSE
                    position = position' - 1
            FI
TRANSITION open_door
  ENTRY           [ TIME : open_dur ]
            ~door_open
          & ~door_moving
          &  ( ~moving
            |  moving
             & EXISTS t: time
                        ( Change ( position, t )
                        & t > Change ( moving )  )  )
          &  ( the_elevator_buttons.floor_requested ( position )
            |  going_up
             &  ( the_floor_buttons [ position ] .up_requested
               |  ~request_above ( position )
                & the_floor_buttons [ position ] .down_requested )
            |  ~going_up
             &  ( the_floor_buttons [ position ] .down_requested
```

```
                |  ~request_below ( position )
                 & the_floor_buttons [ position ] .up_requested )  )
    EXIT
              ~moving
            & door_moving
            & going_up =  ( going_up'
                        &  ( request_above' ( position' )
                          |  the_floor_buttons [ position' ] .up_requested' )
                        |  ~request_below' ( position' )
                         & ~the_floor_buttons [ position' ] .down_requested' )
  TRANSITION close_door
    ENTRY           [ TIME : close_dur ]
              door_open
            & ~door_moving
            & now - t_stop >= Change ( door_open )
    EXIT
              door_moving
  TRANSITION door_stop
    ENTRY           [ TIME : door_stop_dur ]
              door_moving
            & now - t_move_door >= Change ( door_moving )
    EXIT
              ~door_moving
            & door_open = ~door_open'
  END Top_Level
END Elevator

PROCESS SPECIFICATION Elevator_Button_Panel
  LEVEL Top_Level
    IMPORT
            floor, request_dur, clear_dur, the_elevator, the_elevator.position,
            the_elevator.door_open, the_elevator.door_moving
    EXPORT
            floor_requested, request_floor
    VARIABLE
            floor_requested ( floor ) : boolean
    ENVIRONMENT
      /* multiple button pushes should have no effect */
             ( FORALL f: floor
                        ( Change ( floor_requested ( f ) , now )
                        & ~floor_requested ( f )
                     ->   FORALL t: time
                                  ( Start ( request_floor ( f ) )  <= t
                                  & t <= now
                               ->   ~Call ( request_floor ( f ) , t )  )  )  )
      /* requests cannot be made of the elevator to stop at a floor between when the
           door starts opening on that floor until when it starts closing */
            &  ( Change ( the_elevator.door_moving, now )
               & the_elevator.door_moving
               & the_elevator.door_open
            ->   FORALL t: time
                        ( t >= Change [ 2 ]  ( the_elevator.door_moving )
                     ->   ~Call ( request_floor ( the_elevator.position ) , t )  )  )
    INITIAL
            FORALL f: floor
                     ( ~floor_requested ( f )  )
    INVARIANT
      /* buttons only clear after elevator has arrived and started opening the doors */
             ( FORALL f: floor
                        ( Change ( floor_requested ( f ) , now )
                        & ~floor_requested ( f )
```

```
                    ->   EXISTS t: time
                                ( Change [ 2 ]  ( floor_requested ( f )  )  < t
                                & t <= now
                                & past ( the_elevator.position, t )  = f
                                & ~past ( the_elevator.door_open, t )
                                & past ( the_elevator.door_moving, t )  )  )  )
    TRANSITION request_floor ( f: floor )
      ENTRY           [ TIME : request_dur ]
                ~floor_requested ( f )
      EXIT
                floor_requested ( f ) Becomes TRUE
    TRANSITION clear_floor_request
      ENTRY           [ TIME : clear_dur ]
                floor_requested ( the_elevator.position )
              & ~the_elevator.door_open
              & the_elevator.door_moving
      EXIT
                floor_requested ( the_elevator.position ) Becomes FALSE
  END Top_Level
END Elevator_Button_Panel

PROCESS SPECIFICATION Floor_Button_Panel
  LEVEL Top_Level
    IMPORT
              request_dur, clear_dur, the_floor_buttons, the_elevator,
              the_elevator.position, the_elevator.door_open, the_elevator.going_up,
              the_elevator.door_moving, n_floors
    EXPORT
              up_requested, down_requested, request_up, request_down
    VARIABLE
              up_requested, down_requested: boolean
    ENVIRONMENT
      /* multiple button pushes should have no effect */
               ( Change ( up_requested, now )
               & ~up_requested
            ->   FORALL t: time
                           ( Start ( request_up )  <= t
                           & t <= now
                         ->   ~Call ( request_up, t )  )  )
            &  ( Change ( down_requested, now )
               & ~down_requested
            ->   FORALL t: time
                           ( Start ( request_down )  <= t
                           & t <= now
                         ->   ~Call ( request_down, t )  )  )
      /* requests cannot be made of the elevator to stop at a floor between when the
         door starts opening on that floor until when it starts closing */
            &  ( Change ( the_elevator.door_moving, now )
               & the_elevator.door_moving
               & the_elevator.door_open
               & the_floor_buttons [ the_elevator.position ]  = Self
            ->   FORALL t: time
                           ( t >= Change [ 2 ]  ( the_elevator.door_moving )
                         ->    ( past ( the_elevator.going_up, t )
                               ->   ~Call ( request_up, t )  )
                               &  ( past ( ~the_elevator.going_up, t )
                               ->   ~Call ( request_down, t )  )  )  )
    INITIAL
              ~up_requested
            & ~down_requested
    INVARIANT
```

```
  /* buttons only clear after elevator has arrived and started opening the doors */
           ( Change ( up_requested, now )
           & ~up_requested
        ->   EXISTS t: time
                    ( Change [ 2 ]  ( up_requested )  < t
                    & t <= now
                    & the_floor_buttons [ past ( the_elevator.position, t)] = Self
                    & ~past ( the_elevator.door_open, t )
                    & past ( the_elevator.door_moving, t )
                    & past ( the_elevator.going_up, t )  )  )
        &  ( Change ( down_requested, now )
           & ~down_requested
        ->   EXISTS t: time
                    ( Change [ 2 ]  ( down_requested )  < t
                    & t <= now
                    & the_floor_buttons [ past ( the_elevator.position, t)] = Self
                    & ~past ( the_elevator.door_open, t )
                    & past ( the_elevator.door_moving, t )
                    & ~past ( the_elevator.going_up, t )  )  )
  /* the top floor never has an up request and the bottom floor never has a down
       request */
        &  ( the_floor_buttons [ n_floors ]  = Self
        ->   ~up_requested )
        &  ( the_floor_buttons [ 1 ]  = Self
        ->   ~down_requested )
SCHEDULE
  /* calls will be posted within 2 * request_dur time */
           ( Call ( request_up, now - 2 * request_dur )
        ->   EXISTS t: time
                    ( now - 2 * request_dur < t
                    & t <= now
                    & past ( Change ( up_requested, t ) , t )
                    & past ( up_requested, t )  )  )
        &  ( Call ( request_down, now - 2 * request_dur )
        ->   EXISTS t: time
                    ( now - 2 * request_dur < t
                    & t <= now
                    & past ( Change ( down_requested, t ) , t )
                    & past ( down_requested, t )  )  )
TRANSITION request_up
  ENTRY           [ TIME : request_dur ]
            ~up_requested
          & the_floor_buttons [ n_floors ]  ~= Self
  EXIT
            up_requested
TRANSITION request_down
  ENTRY           [ TIME : request_dur ]
            ~down_requested
          & the_floor_buttons [ 1 ]  ~= Self
  EXIT
            down_requested
TRANSITION clear_up_request
  ENTRY           [ TIME : clear_dur ]
            up_requested
          & the_floor_buttons [ the_elevator.position ]  = Self
          & the_elevator.going_up
          & ~the_elevator.door_open
          & the_elevator.door_moving
  EXIT
            ~up_requested
TRANSITION clear_down_request
```

```
        ENTRY           [ TIME : clear_dur ]
                      down_requested
                    & the_floor_buttons [ the_elevator.position ]  = Self
                    & ~the_elevator.going_up
                    & ~the_elevator.door_open
                    & the_elevator.door_moving
        EXIT
                      ~down_requested
    END Top_Level
  END Floor_Button_Panel
END Elevator_System
```

References

[1] A. Alborghetti, A. Gargantini, A. Morzenti, Providing automated support to deductive analysis of time critical systems, Proc. 6th European Software Engineering Conf., 1997.

[2] M. Archer, C. Heitmeyer, Mechanical verification of timed automata: a case study, Proc. Real-Time Technology and Applications Symp., 1996, pp. 192–203.

[3] M. Archer, C. Heitmeyer, Human-style theorem proving using PVS, Proc. 10th Internat. Conf. on Theorem Proving in Higher Order Logics, August 1997, pp. 33–48.

[4] R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, J. Van Tassel, Experience with embedding hardware description languages in HOL, Proc. Internat. Conf. on Theorem Provers in Circuit Design, 1992, pp. 129–156.

[5] L. Bun, Checking properties of ASTRAL specifications with PVS, Proc. 2nd Ann. Conf. of the Advanced School for Computing and Imaging, 1996, pp. 102–107.

[6] L. Bun, Embedding Astral in PVS, Proc. 3rd Ann. Conf. of the Advanced School for Computing and Imaging, 1997, pp. 130–136.

[7] A. Coen-Porisini, C. Ghezzi, R.A. Kemmerer, Specification of realtime systems using ASTRAL, IEEE Trans. Software Eng. 23 (9) (1997) 572–598.

[8] A. Coen-Porisini, R.A. Kemmerer, D. Mandrioli, A formal framework for ASTRAL intralevel proof obligations, IEEE Trans. Software Eng. 20 (8) (1994) 548–561.

[9] A. Coen-Porisini, R.A. Kemmerer, D. Mandrioli, A formal framework for ASTRAL inter-level proof obligations, Proc. 5th European Software Engineering Conf., 1995, pp. 90–108.

[10] J. Crow, S. Owre, J. Rushby, N. Shankar, M. Srivas. A tutorial introduction to PVS, Workshop on Industrial-Strength Formal Specification Techniques, 1995.

[11] R.E. Filman, D.P. Friedman, Coordinated Computing: Tools and Techniques for Distributed Software, McGraw-Hill, New York, 1984.

[12] C. Ghezzi, R.A. Kemmerer, ASTRAL: an assertion language for specifying realtime systems, Proc. 3rd European Software Engineering Conf., 1991, pp. 122–140.

[13] C. Ghezzi, R.A. Kemmerer, Executing formal specifications: the ASTRAL to TRIO translation approach, Proc. Symp. on Testing, Analysis, and Verification, 1991.

[14] M. Gordon, Notes on PVS from a HOL perspective, Available at <http://www.cl.cam.ac.uk/users/mjcg/PVS.html>, 1995.

[15] M.J.C. Gordon, T.F. Melham (Eds.), Introduction to HOL: A Theorem Proving Environment for Higher Order Logic, Cambridge University Press, Cambridge, 1993.

[16] R. Hale, R. Cardell-Oliver, J. Herbert, An embedding of timed transition systems in HOL, Formal Methods System Des. 3 (1–2) (1993) 151–174.

[17] C. Heitmeyer, N. Lynch, The generalized railroad crossing: a case study in formal verification of real-time systems, Proc. 15th Real-Time Systems Symp., 1994, pp. 120–131.

[18] C. Heitmeyer, D. Mandrioli (Eds.), Formal Methods for Real-Time Computing, Wiley, New York, 1996.

[19] M. Kaufmann, J. Strother Moore, ACL2: an industrial strength version of Nqthm, Proc. 11th Ann. Conf. on Computer Assurance, 1996, pp. 23–34.

[20] P.Z. Kolano, The ASTRAL Specifications of 8 Real-Time Systems. Tech. Report TRCS99-08, Computer Science Dept., University of California, Santa Barbara, 1999.

[21] P.Z. Kolano, Tools and techniques for the design and systematic analysis of real-time systems, Ph.D. Thesis, University of California, Santa Barbara, 1999.
[22] P.Z. Kolano, Z. Dang, R.A. Kemmerer, The design and analysis of real-time systems using the ASTRAL software development environment, Ann. Software Eng. 7 (1999) 177–210.
[23] P.Z. Kolano, R.A. Kemmerer. Classification schemes to assist in the analysis of real-time systems, Proc. Internat. Symp. on Software Testing and Analysis, 2000, pp. 86–95.
[24] P.Z. Kolano, R.A. Kemmerer, D. Mandrioli, Parallel refinement mechanisms for real-time systems, Proc. 3rd Internat. Conf. on Fundamental Approaches to Software Engineering, 2000, pp. 35–50.
[25] L. Lamport, A new solution of Dijkstra's concurrent programming problem, Comm. ACM 17 (8) (1974) 453–455.
[26] C. Lewerentz, T. Lindner (Eds.), Formal Development of Reactive Systems: Case Study Production Cell, Springer, New York, 1995.
[27] Official 1996 Olympic Web Site, <http://www.atlanta.olympic.org>.
[28] J.U. Skakkebaek, N. Shankar, Towards a duration calculus proof assistant in PVS, 3rd Internat. Symp. on Formal Techniques in Real-Time and Fault-Tolerant Systems, 1994, pp. 660–679.
[29] J.M. Spivey, Specifying a real-time kernel, IEEE Software 7 (5) (1990) 21–28.
[30] P.T. Ward, S.J. Mellor, Structured development for real-time systems, Yourdon Press, New York, 1985.
[31] W.D. Young, Comparing verification systems: interactive consistency in ACL2, Proc. 11th Annual Conf. on Computer Assurance, 1996, pp. 35–45.